A Lagrangian Approach Based on the Natural Neighbor Interpolation

Facundo del Pin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Lagrangian Approach Based on the Natural Neighbor Interpolation

Facundo DEL PIN

N° 4090

Novembre 2000

THÈME 4

ISSN 0249-6399 ISRN INRIA/RR--4090--FR+ENG

INRIA
SOPHIA ANTIPOLIS

A Lagrangian Approach Based on the Natural Neighbor Interpolation

Facundo DEL PIN *

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet SINUS

Rapport de recherche n° 4090 — Novembre 2000 — 115 pages

Abstract: A Lagrangian formulation is constructed using the so-called Moving-Particle Semi-Implicit Method. The approximation scheme is then modified to incorporate the natural neighbor interpolation in the definition of the shape functions. In this way, a multi-scale method results. This method is presented as an alternative for solving partial-differential equations (PDE). Second-order convergence is achieved when uniform rectangular (finite-difference-type) grids are used. Laplace and Poisson equations are solved in somewhat more general geometries as examples; the method demonstrates the same second-order accuracy as classical finite-elements (FEM), but allows a different treatment of geometry potentially powerful for local mesh adaptation in particular.

Key-words: Grid-less, Particle Method, Interpolation, Voronoi diagrams, Delaunay triangulation, Natural neighbour.

* Université Pierre et Marie CURIE, PARIS VI

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

A Lagrangian Approach Based on the Natural Neighbor Interpolation

Résumé : Une approche lagragienne est construite sur la base de la méthode particulaire semi-implicite. Le schéma d'approximation est ensuite modifié pour inclure l'interpolation par les voisins naturels dans la définition des fonctions de forme. Il en résulte ainsi une méthode multiéchelle, alternative intéressante pour résoudre des équations aux dérivées partielles (EDP). Lorsque le maillage est rectangulaire et uniforme, de type différences-finies, on obtient une précision du second ordre. A titre d'exemples, on résout les équations de Laplace et de Poisson sur des géométries un peu plus générales et on observe que la méthode conserve la même précision que les éléments finis du second-ordre, tout en autorisant un traitement différent de la géométrie, potentiellement puissant pour l'adaptation locale du maillage en particulier.

Mots-clés : Mesh-less, Interpolation, Natural Neighbor, Voronoi Diagrams, Delunay Triangulation

ACKNOWLEDGMENTS

I am very grateful to the numerous collegues who have taken part in this work, providing me with information, tools, ideas that reveled essential to carry out this report:

Jean-Daniel Boissonnat

Frank Da

Jean-Antoine Desideri

Frédéric Cazals

Ales Janka

Patricia Maleyran

Stephane Walles

Contents

List of Figures

1 Introduction

Computer simulation has been required for analysing increasingly complex problems from the geometrical or phisical modelling standpoint. However there are many problems that are still difficult to analyze via computer simulation with either the finite volume or the finite element methods.

The Finite Element Method (FEM) is a well-established numerical method which has been applied in different fields of engeneering and applied sciences. In spite of its numerous advantages, certain class of problems, such as crack growth, plate bending, large deformations, higher order partial differential equations (PDE's), moving phase boundaries, and modeling of multiscale phenomena are still difficult to analyze via FEM.

To this end, the emergence of the mesh-less methods, has engendered significant interest and is viewed as a step in the right direction.

Computer simulation using mesh-less methods has the capability to analyze more complex geometry and physics than grid methods. Particularly, topological deformation of the fluid can be analyzed by mesh-less methods, while it is difficult to fit and move a grid continuously in such domains. In addition, grid generation and topological relations among the particles are not necessary.

The method chosen in this analysis is the *moving-particle semi-implicit method* (MPS)[1]. This method discretizes the continuous media using particles. The particles interact among them by a function called *kernel function* which relates a particle with all the particles in its vicinity. The partial differential equations, such as the Navier Stokes equations are discretized by particle interaction models according to the kernel function. As we can see, to chose an appropriate kernel function is vital to approximate the problem satisfactorily. Very often this function depends on the distances among particles and their interaction radii are bounded. Thus a particle interacts with a limited amount of particles in its vicinity. This is satisfactory when the storage memory takes is a critical parameter. On the other hand, it is a problem to determine how many particles are enough for a good interpolation and further, how to chose the interaction radius to get the necessary number of particles for interpolation in a given problem. In a general situation this problem may have different density of points in the same geometry (for example mesh refinement). Then different radii should be used for each particle to keep the amount of particles desired for the approximation. This creates a more complicated

problem when the particles are moving and their positions change and thus a new radius has to be computed. Another difficulty associated with this kind of interpolation is the necessity to find the neighboring particles of a given particle, necessitating a costly test with all the other particles to determine which of them are inside the interpolation radius.

If we compare some characteristics of the FEM to most mesh-less methods, the local interpolant in FEM is constructed on the basis of an element structure while in most mesh-less methods the local approximation is based on weight functions of compact support. The interpolation using Natural Neighbors uses a normalized Lebesgue measure that takes into account the distribution of nodes and their relation to each other in space. The FEM, as a Galerkin method, possesses the properties of linear completeness which is a sufficient condition for convergence in the application to second-order elliptic PDEs. The distinction lies in the manner in which the linear completeness conditions are obtained. In mesh-less methods, the approximation spaces covers the continuity of the weight function for smooth problems and hence higher-order continuous (C^k, $k > 0$) functions of interpolation are already constructed [4]. Traditional FEM are C^0 while the approximation by natural neighbors is smooth everywhere except at the nodes where it is C^0. The local approximation spaces in finite elements are uniformly independent [3]. On the other hand, irregular arrangements in mesh-less approximations such as MPS do result in marked deterioriation in the properties of approximation space. This is so since isotropic weight functions are noticeably biased toward regions of higher nodal density, at the expense of equi-directionality which is contended to be an essential ingredient for good approximability. The density and spatial location of nodes are taken into account in the construction of the natural neighbor interpolant. This allows for a natural means to construct stable approximation spaces that are based on irregular nodal arrangements. In this work, the Delaunay triangles which are the dual of the Voronoi diagram, are used in the numerical computation of the interpolation. However, unlike the finite element method where angle restrictions are imposed on the Delaunay triangles for the convergence of the method[5]. No such constraints are imposed on the shape, size, and angles of the Delaunay triangles in the interpolation by natural neighbors.

The reasons above were the motivation to attempt to adapt a new kind of interpolation, independent of the problem's physical scale, to this grid-less method. The interpolation chosen was the NNI which is going to be described in detail later. A different way to approximate

partial differential equations was obtained. When the comparison of the solutions were carried out, the NNI approximation showed the same trend of convergence as the FEM and the Finite Difference methods. Adding this to the advantages mentioned previously the method proved a good alternative for solving PDEs and revealed to be very flexible with respect to geometry adaptation, implementation and applications.

The C^{++} *CGAL* library [6] was used to compute the interpolation. The results were visualized using Vigie [7].

The outline of this report follows.

In Chapter 2, the Moving-Particle Semi-Implicit Method is introduced. We present how the laplacian and the gradient are approximated in a particle discretization.

In Chapter 3 an empirical measure of the error is measured for the MPS method by solving a collapse water-column.

In Chapter 4 an introduction of the geometrical principles needed to solve the interpolation by natural neighbors is presented. Voronoi diagrams, Delaunay tessellation and their geometric properties are explained as an introduction to the formal presentation of the interpolation.

In Chapter 5, the natural neighbor interpolation and its associated shape function are explained in detail and some special cases are contemplated.

Chapter 6 explains the computation of the geometry introduced in Chapter 4. The *CGAL* library is introduced as well as some methods to compute the interpolation.

In Chapter 7, the incorporation of the natural neighbor interpolation in the particle method is carried out. The order of convergence in the case of a uniform rectangular grid is assessed.

In Chapter 8, the numerical results using the model described in Chapter 7 are presented. They are compared with other methods such as finite elements and finite difference.

Chapter 9 collects a summary of the results obtained in this work and some perspectives for this method.

2 Moving-Particle Semi-Implicit Method

In order to approximate the Navier Stokes equation for incompressible flow, the MPS method is introduced as a Lagrangian formulation. This means that the modeling does not depend on a mesh and the particles move following the motion of the flow. In this way, the convection term in the Navier Stokes equation is calculated without numerical diffusion. Numerical techniques that require a fixed Eulerian grid run into difficulties when the convection displacement of the diffusing quantity greatly exceeds the diffusive displacement. In this situation it is hard to provide sufficient mesh refinement where it is needed, while the proper direction of discretization is not known a priori to the solution and may result in excessive numerical diffusion [11]. To overcome some of these difficulties, MPS allows the particles to move, following the distribution of gradient.

The governing equations are introduced first and then approximated.

2.1 Governing Equations

Governing equations for incompressible flows are the continuity and the momentum equations:

$$\frac{\partial \rho}{\partial t} = 0 \tag{1}$$

and

$$\frac{Du}{Dt} = -\frac{1}{\rho}\nabla P + f \tag{2}$$

The continuity equation is written with density, while velocity divergence is usually used in grid method. The left-hand side of Eq. 2 involves the Lagrangian derivatives including convection terms. This is directly calculated by tracing particle motion. The right-hand side consists of pressure gradient and external force terms. All terms expressed by differential operators are to be replaced by particles interactions.

2.2 Particle Interaction Models

2.2.1 Kernel

A particle interacts with others in its vicinity, and this is described by means of the *Kernel Function* $w(r)$, where r is the distance between two particles. An example of *Kernel Function* is given by (Fig. 1):

$$w(r) = \begin{cases} \frac{r_e}{r} - 1 & if \quad (0 \leqq r < r_e) \\ \\ 0 & if \quad (r_e \leqq r) \end{cases} \tag{3}$$

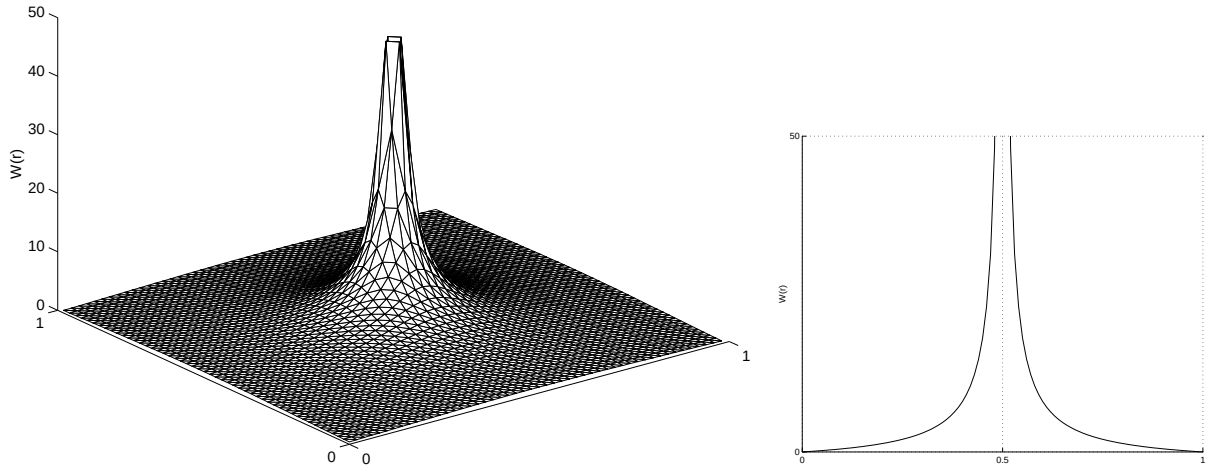

Figure 1: The Kernel Function shown in Eq. 3 .

The radius of the interaction is determined by the parameter r_e, see Fig. 2. Since the area that is covered by this kernel function is bounded, a particle interacts with a finite number of neighboring particles

2.2.2 Particle Number Density

The particle number density at coordinate $\mathbf{r}_i$ where particle i is located is defined by

$$\langle n \rangle_i = \sum_{j \neq i} w(|\mathbf{r}_j - \mathbf{r}_i|) \tag{4}$$

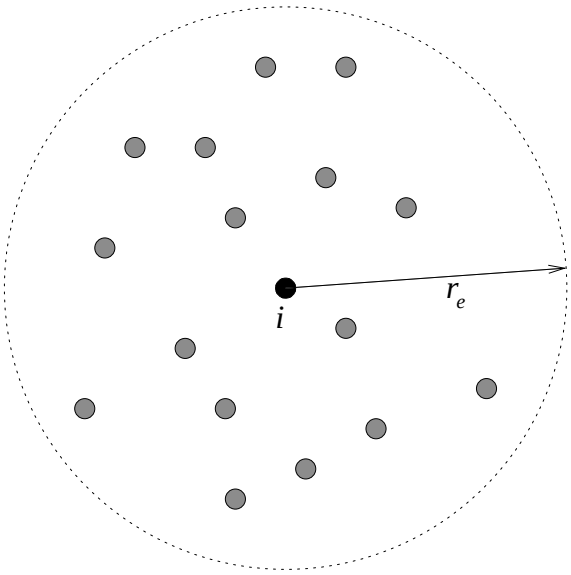

Figure 2: Interaction between particles.

In this equation, the contribution from particle i itself is not considered. When the number of particles in a unit volume is denoted by $\langle N \rangle_i$ it follows that

$$\langle N \rangle_i = \frac{\langle n \rangle_i}{\int_V w(r) dv} \tag{5}$$

The denominator of Eq. 5 is the integral of the kernel in the whole region, excluding a central part occupied by particle i. Assuming that the particles have the same mass m, we can see that the fluid density is proportional to the particle number density:

$$\langle \rho \rangle_i = m \langle N \rangle_i = \frac{m \langle n \rangle_i}{\int_V w(r) dv} \tag{6}$$

Thus, it is legitimate to say that the continuity equation is satisfied if the particle number density is constant. This constant value is denoted by n^0.

2.2.3 Modeling of Gradient

A gradient vector between two particles i and j possessing scalar quantities ϕ_i and ϕ_j at coordinates $\mathbf{r}_i$ and $\mathbf{r}_j$ is simply defined by

$$\frac{(\phi_j - \phi_i)(\mathbf{r}_j - \mathbf{r}_i)}{|\mathbf{r}_j - \mathbf{r}_i|^2} \tag{7}$$

It is possible to evaluate the gradient vector with any combination of two particles. The gradient vectors between particle i and its neighboring particles j are weighed with the kernel function and averaged to obtain a gradient vector at particle i:

$$\langle \nabla \phi \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \Big[\frac{\phi_j - \phi_i}{|\mathbf{r}_j - \mathbf{r}_i|^2} (\mathbf{r}_j - \mathbf{r}_i)\, w(|\mathbf{r}_j - \mathbf{r}_i|) \Big] \tag{8}$$

where d is the number of space dimensions (2 in 2 dimensions). This model is applied to the pressure gradient term in MPS method.

The current model is not sensitive to absolute pressure. This is consistent with the property of incompressible fluids, which depends on the relative pressure distribution.

2.2.4 Modeling of Laplacian

Let us consider a time-dependent diffusion problem involving the unknown function ϕ:

$$\frac{d\phi}{dt} = \nu \nabla^2 \phi \tag{9}$$

where ν is the diffusion coefficient.

During a time interval Δt, the variance of the distribution of ϕ increases by

$$\Sigma = 2d\nu\Delta t \tag{10}$$

where d is the number of space dimensions. In the current model, part of quantity ϕ_i of particle i is distributed to the neighboring particles according to the kernel function such that the variance increase is equal to $2d\nu\Delta t$. Thus the quantity transfered from particle i to j is

$$\nabla \phi_{i \to j} = \frac{2d\nu\Delta t}{n^0 \lambda} \phi_i w(|\mathbf{r}_j - \mathbf{r}_i|) \tag{11}$$

where

$$\lambda = \frac{\int_V w(r) r^2\, dv}{\int_V w(r)\, dv} \tag{12}$$

As far as linear diffusion is concerned, the quantity transfer can be superimposed. As a result, a natural way to approximate the laplacian is given by [2]

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{n^0 \lambda} \sum_{j \neq i} (\phi_j - \phi_i)\, w(|\mathbf{r}_j - \mathbf{r}_i|) \tag{13}$$

In the case $f_i = 1$ and $f_j = 0$, the variance increase in Eq. 9 during Δt is

$$\Sigma = \frac{2\Delta t \nu d}{\lambda n^0} \sum_{j \neq i} [|\mathbf{r}_j - \mathbf{r}_i|^2 (\phi_j - \phi_i) w(|\mathbf{r}_j - \mathbf{r}_i|)] \tag{14}$$

that could be written as

$$\Sigma = \frac{2\Delta t \nu d}{\lambda n^0} \sum_{j \neq i} [|\mathbf{r}_j - \mathbf{r}_i|^2 w(|\mathbf{r}_j - \mathbf{r}_i|)] \tag{15}$$

Comparing Eq. 15 and Eq. 10, we have

$$2\nu d \Delta t = \frac{2\Delta t d \nu}{\lambda n^0} \sum_{j \neq i} [|\mathbf{r}_j - \mathbf{r}_i|^2 w(|\mathbf{r}_j - \mathbf{r}_i|)] \tag{16}$$

where by canceling the common terms we arrive to the next equation

$$1 = \frac{\sum_{j \neq i} [|\mathbf{r}_j - \mathbf{r}_i|^2 w(|\mathbf{r}_j - \mathbf{r}_i|)]}{\lambda n^0} \tag{17}$$

where n^0 is given by Eq. 4. Thus λ can be approximated by

$$\lambda = \frac{\sum_{j \neq i} [|\mathbf{r}_j - \mathbf{r}_i|^2 w(|\mathbf{r}_j - \mathbf{r}_i|)]}{\sum_{j \neq i} w(|\mathbf{r}_j - \mathbf{r}_i|)} \tag{18}$$

The current model of Laplacian is conservative since the quantity lost by particle i is exactly received by particle j.

2.2.5 Modeling Incompressibility

The continuity equation requires that the fluid density should be constant. This is equivalent to the particle number density being constant, n^0. When the particle number density n^* is not n^0, it is implicitly corrected to n^0 by

$$n^* + n\prime = n^0 \tag{19}$$

where $n\prime$ is the correction value. This correction is related to the velocity correction $\mathbf{u}\prime$ through the mass conservation equation:

$$\frac{1}{\Delta t} \frac{n\prime}{n^0} = -\nabla \cdot \mathbf{u}\prime \tag{20}$$

The velocity correction value is derived from the implicit pressure gradient term as follows

$$\mathbf{u}\prime = -\frac{\Delta t}{\rho}\nabla P^{n+1} \tag{21}$$

With Eqs. 19, 20 and 21 a Poisson equation for pressure is obtained:

$$\langle \nabla^2 P^{n+1} \rangle_i = -\frac{\rho}{\Delta t^2}\frac{\langle n^* \rangle_i - n^0}{n^0} \tag{22}$$

The right side represents by a deviation of the particle number density from the constant value, while usually velocity divergence is used in grid methods. The left-hand side of Eq. 22 is discretized by the Laplacian model, Eq. 13. Finally we have simultaneous equations expressed by a linear symmetric matrix. The pressure gradient terms are calculated from the gradient model, Eq. 8, where scalar ϕ is substituted by P^{n+1}.

Eq. 22 is obtained through the manipulation of a differential equation so that the discretization forms of the original equations (Eqs. 19, 20, 21) are not conservative. In other words, the modified particle number density, which is evaluated from the modified coordinates of particles, is not equal to n^0. However this inconsistency is not serious because the error of the particle number density is not accumulated in each time step and since the mass conservation is strictly guaranteed by keeping the number of particles.

The overall algorithm is described in Fig. 3. In each time step, source terms are explicitly calculated and temporal velocities $\mathbf{u}_i^*$ are obtained. Next, the motion of particles is calculated, and temporal coordinates $\mathbf{r}_i^*$ are obtained. This corresponds to the convection term. The Poisson equation for pressure is solved with the source term representing the deviation of the particle number density, and the new-time pressure values are obtained. Finally we have new-time velocities and coordinates by adding the pressure gradient terms with the new-time pressure values. This algorithm is basically semi-implicit time marching.

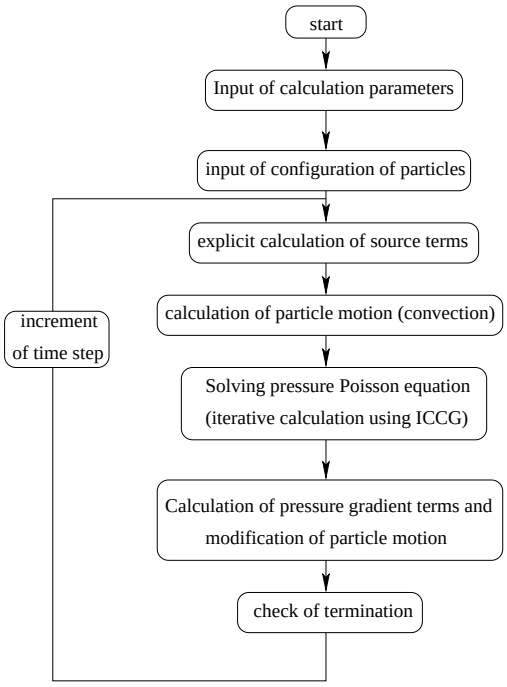

Figure 3: Calculation Algorithm of MPS.

3 Empirical Measure of the Order of Approximation in MPS

As there was no information about the order of approximation of the error in the MPS method, an empirical way to measure it was applied.

As it is well known a discrete system is a good approximation to the continuous system if it gets closer to the exact solution when the number of discret elements increases. This means that the approximation converges. So, when increasing the number of discret elements, we should observe a trend in the approximated solution converging to the exact solution. The order of the approximation gives the rate with which the method converges when we add elements to the discretization.

When the method used for the approximation has others parameters but not only the number of discret elements, the approximation gets more complicated and the convergence does not depend only on the number of discret elements. For example in the MPS method the interpolation radius in Eq. 3 has a very important role and a satisfactory solution is only acheeved when an appropriate radius is used.

There is no explicit way to do this when Eq. 3 is used the interpolation. When the empirical measure of the error was measured a fixed typical radius was chosen and the number of particles was increased. To do this, the water-column collapse problem was solved (Fig. 4)[1] and we measured the impulse on the bottom of the container.

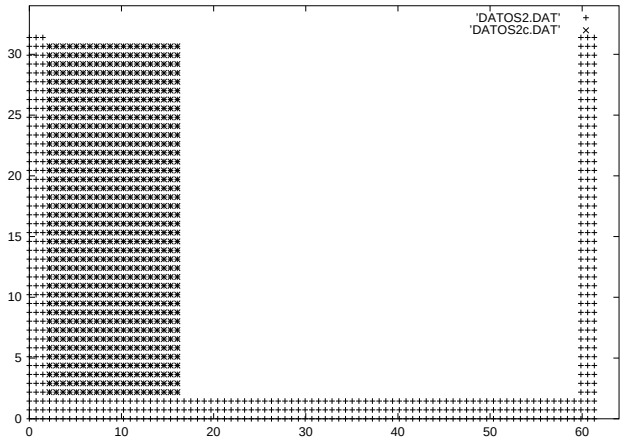

Figure 4: Collapse of a water column.

The impulse is defined as

$$I = \int_0^T F\, dt \approx \sum_{i=0}^{n} F_i\, \Delta t_i \tag{23}$$

where n is the number of time steps and F is the force on the bottom of the container simply defined as

$$F = \int_A P\, dA \approx \sum_i P_i\, \Delta A_i \tag{24}$$

and P is the pressure on the bottom of the container obtained by the Eq. 22.

So let us suppose that

$$I_N = I + C\Big(\frac{1}{N}\Big)^{\alpha} + \dots \tag{25}$$

where :

1. I is the impulse (Eq. 23);
2. I_N is the approximate value for the impulse;
3. C is a constant;
4. N is the number of particles used to solve the problem;
5. and α is the empirical order of the method.

If we take N for different cases i such that $N_{i+1} = 2N_i$ we can write

$$\frac{I_{500} - I_{1000}}{I_{1000} - I_{2000}} = \frac{1 - \frac{1}{2^\alpha}}{\frac{1}{2^\alpha} - \frac{1}{4^\alpha}} = 2^\alpha \tag{26}$$

thus

$$\alpha = \log_2\Big(\frac{I_{500} - I_{1000}}{I_{1000} - I_{2000}}\Big) \tag{27}$$

where α is the experimental order of the method.

In Fig. 5 we can see in a logarithmic scale the results for different cases, choosing different radii of interpolation.

While the number of particles increases it is not possible to see a trend. This is due in part to the fact that there is too large a number of varying parameters and that sometimes the values of these parameters are not very well defined. This is the case for the radius of

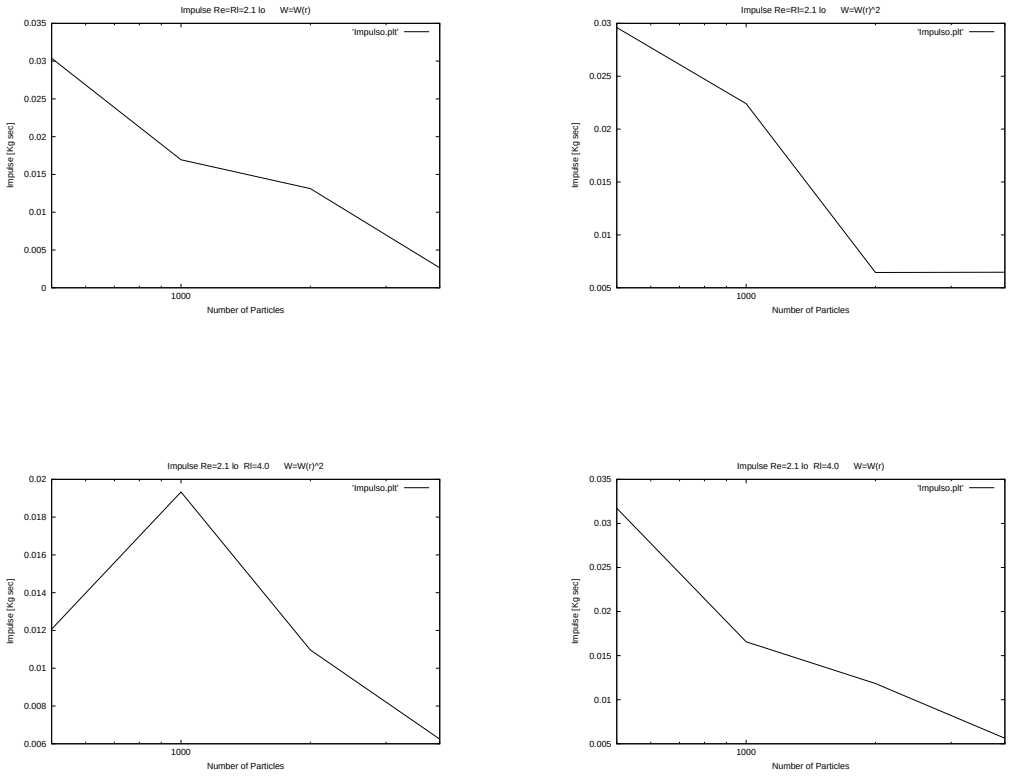

Figure 5: Impulse vs. Number of particles in logarithmic scale scale.

interaction in the *kernel function*. This is another reason that motivated the application of a multi-scale interpolation.

After this series of preliminary tests, we conclude that the regular method MPS /citemps lacked generality and depend on several parameters to be defined somewhat arbitrarily. This motivated us to look for a more general alternative, and to propese a kernel function defined by means of the natural-neighbor interpolation. Before making this methodological change, we review in the next section, certain classical geometrical concepts necessary subsequently

4 Voronoi Diagram and Delaunay Tessellation

The Voronoi diagram and its dual Delaunay tessellation are one of the most fundamental and useful geometrical constructs that define an irregular set of points. For simplicity, and keeping in mind the applications that are persued in this work, we consider two-dimensional Euclidean space $\Re^2$; the theory, however, is applicable in a general d-dimensional framework. Consider a set of distinct nodes $N = n_1, n_2, \ldots, n_m$ in $\Re^2$. The Voronoi diagram (or first-order Voronoi diagram) of the set N is a subdivision of the plane into regions T_I (closed and convex, or unbounded), where each region T_I is associated with a node n_I, such that any point in T_I is closer to n_I (nearest neighbor) than to any other node $n_J \in N$ $(J \neq I)$: T_I is the locus of points closer to n_I than to any other node. The regions T_I are the Voronoi cells of n_I. Thus, the Voronoi polygon T_I is defined as

$$T_I = \{\mathbf{x} \in \Re^2 : d(\mathbf{x}, \mathbf{x_I}) < d(\mathbf{x}, \mathbf{x_J}) \forall\, J \neq I\} \tag{28}$$

where $d(\mathbf{x}_J, \mathbf{x_I})$, the Euclidean distance between $\mathbf{x_I}$ and $\mathbf{x_J}$. The Voronoi diagram for node A for a set N consisting of seven nodes is shown in Fig. 7.In Fig. 6 it is seen that each Voronoi cell T_I is the intersection of finitely many open half-spaces, each being delimited by the perpendicular bisector (hyperplane in $\Re^d$) of the line joining nodes n_I and n_J $(J \neq I)$. Consequently, for all nodes n_I that are inside the convex hull [1], the Voronoi cells are closed and convex, while the cells associated with nodes on the boundary of the convex hull are unbounded.

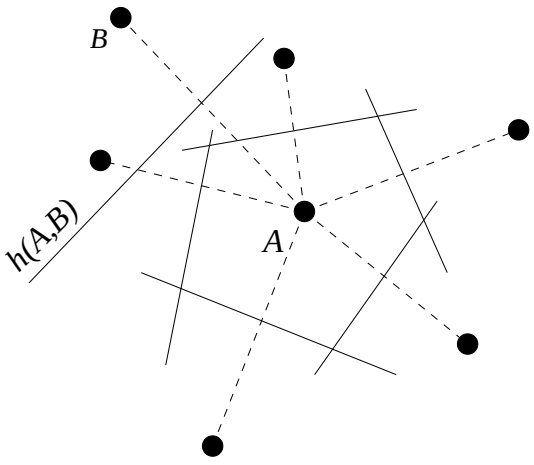

Figure 6: Voronoi cell for point A.

[1]The convex hull $CH(N)$ of the set of nodes N is the smallest convex set containing N.

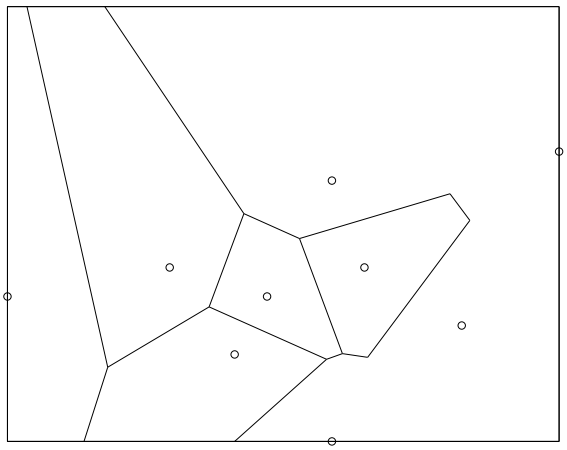

Figure 7: Voronoi Diagram.

The nearest neighbor problem and many of its variants in computational geometry are prototypical examples that illustrate the use of Voronoi diagram. However, the implications of Voronoi diagram are far-reaching, with many applications in the natural sciences, physical sciences, and engeneering.

The Delaunay triangulation, which is the straight dual of the Voronoi diagram, is constructed by connecting the nodes whose Voronoi cells have common boundaries (Fig. 8). The duality between the two implies that there is a Delaunay edge between two nodes in the plane if and only if their Voronoi cells share a common edge. Among all triangles, the Delaunay triangles maximize the minimum angle. Another important property of Delaunay triangulation is the *empty circumcircle criterion*. If $DT(n_J, n_K, n_L)$ is any Delaunay triangle of the nodal set N, then the circumcircle of DT contains no other nodes of N. In the context of natural neighbor interpolation, these circles are known as natural neighbor circumcircle. The center of the natural neighbor circumcircle is a vertex of the Voronoi cell. If the nodal set N is such that only three nodes lie on the circumcircle of any Delaunay triangle (non-degenerate case), then precisely three edges meet to form a Voronoi vertex. In Fig. 9, the natural neighbor circumcircles and the associated Delaunay triangulation are shown.

From an algorithmic viewpoint, since the Voronoi diagram and the Delaunay triangulation share a common bond (duality), the combinatorial structure of either structure is completely determined from its dual. The Voronoi diagram in $\Re^d$ is also closely related to the convex hull in $\Re^{d+1}$, which is also the basis for computing the Voronoi diagram in $\Re^d$.

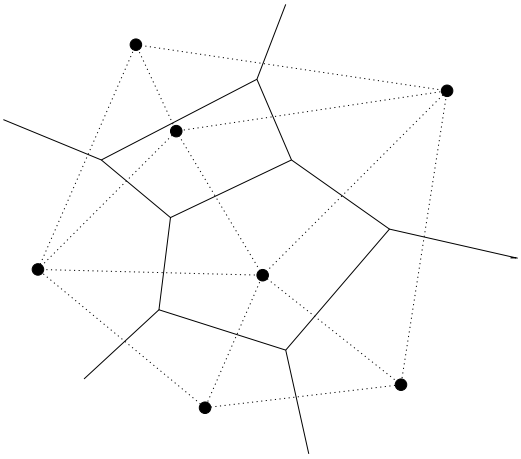

Figure 8: Delaunay Triangulation.

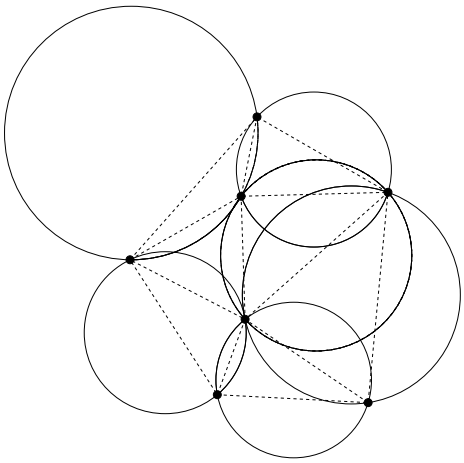

Figure 9: Natural neighbor circumcircles.

4.1 Construction

Natural neighbor coordinates were introduced by Sibson (1980) as a means for data interpolation and smoothing. The concept of nearest neighbors and neighboring nodes is embedded in the first-order Voronoi diagram. By a similar extension, one can construct higher order ($k-order$, $k>1$) Voronoi diagrams in the plane. Of particular interest in the context of natural neighbor coordinates is the case $k=2$, which is the second-order Voronoi diagram . The second-order Voronoi diagram of the set of nodes N is a subdivision of the plane into cells T_{IJ}, where each region T_{IJ} is associated with a nodal-neighbor-pair (n_I, n_J) (k-tuple for the k-order Voronoi diagram), such that T_{IJ} is the locus of all points that have n_I as the nearest neighbor, and n_J as the second nearest neighbor. It is emphasized that the cell T_{IJ} is non-empty if and only if n_I and n_J are neighbors. The second-order Voronoi cell T_{IJ} $(I \neq J)$ is defined as:

$$T_{IJ} = \{\mathbf{x} \in \Re^2 \,:\, d(\mathbf{x}, \mathbf{x_I}) < d(\mathbf{x}, \mathbf{x_J}) < d(\mathbf{x}, \mathbf{x_K}) \,\forall\, K \neq I, J\} \tag{29}$$

In order to quantify the neighbor relation for any point $\mathbf{x}$ introduced into the tessellation, Sibson used the concept of second-order Voronoi cells, and thereby introduced natural neighbors and natural neighbor coordinates. The notion of neighboring node is broadened and generalized to yield a new measure of "neighborlines" by the definition of natural neighbors. In Fig. 10, a point $\mathbf{x}$ is introduced into the Voronoi diagram of the set N discussed in Section 4.

If $\mathbf{x}$ is tessellated along with the set of nodes N, then the natural neighbors of $\mathbf{x}$ are those nodes which form an edge of a triangle with $\mathbf{x}$ in the new triangulation. Using the empty circumcircle criterion, we arrive at the result that if $\mathbf{x}$ lies within the circumcircle of a triangle $DT(n_I, n_J, n_K)$, then n_I, n_J and n_K are its natural neighbors. In Fig. 11, the perpendicular bisectors from point $\mathbf{x}$ to its natural neighbors are constructed and the Voronoi cell $T_{\mathbf{x}}$ (closed polygon *abcd*) is obtained. It is observed that $\mathbf{x}$ has four ($n=4$) natural neighbors, namely nodes 1, 2, 3 and 4.

Let $k(\mathbf{x})$ be a Lebesgue measure (length, area or volume in 1D, 2D or 3D respectively) of $T_{\mathbf{x}}$, and $k(\mathbf{x})$ $(I = 1-4)$ be that of $T_{\mathbf{x}I}$. In two-dimensions, the measures are areas, and hence we denote $A(\mathbf{x}) \equiv k(\mathbf{x})$ and $A_I(\mathbf{x}) \equiv k_I(\mathbf{x})$. The natural neighbor coordinates of $\mathbf{x}$ with respect to a natural neighbor I is defined as the ratio of the area of overlap of the Voronoi cells T_I and

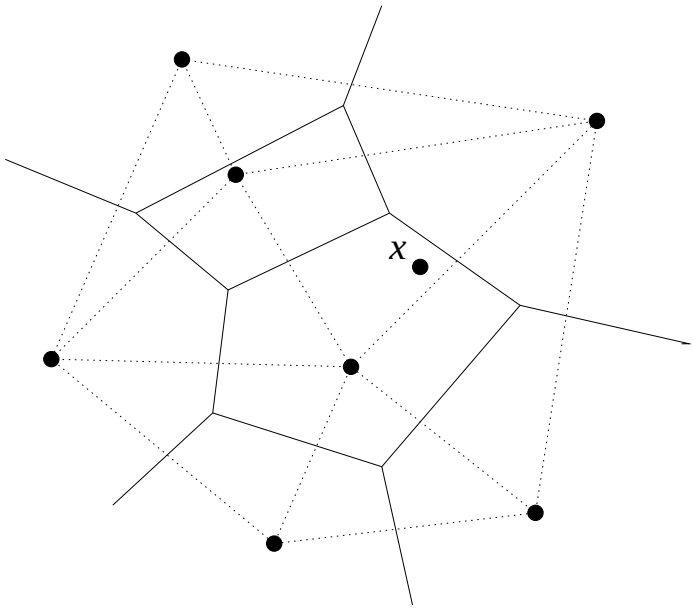

Figure 10: Original Voronoi diagram and **x**.

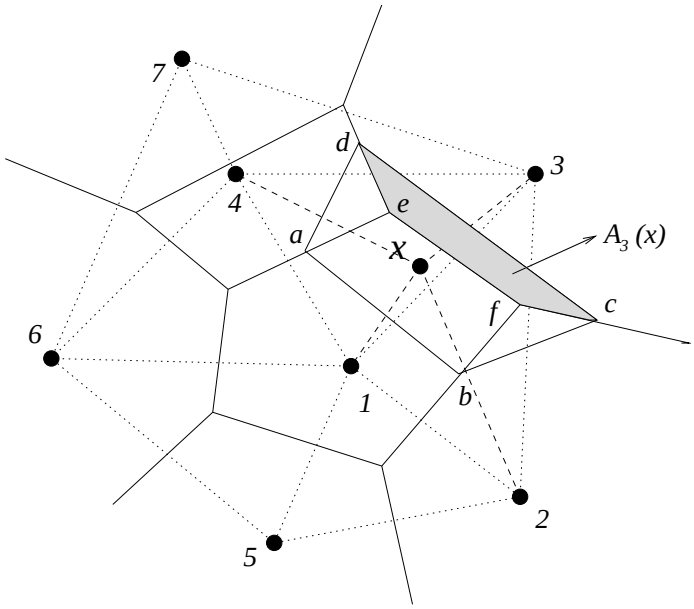

Figure 11: First and second order Voronoi cells about **x**.

$T_{\mathbf{x}}$ to the total area of the Voronoi cell of $\mathbf{x}$:

$$\Phi_I(\mathbf{x}) = \frac{A_I(\mathbf{x})}{A(\mathbf{x})} \tag{30}$$

where I ranges from 1 to n, and $A(\mathbf{x}) = \sum_{J=1}^{n} A_J(\mathbf{x})$. The four regions shown in Fig. 11 are the second-order cells, while their union (closed polygon *abcd*) is a first order Voronoi cell. In Fig. 11 the shape function $\Phi_3(\mathbf{x})$ is given by

$$\Phi_3(\mathbf{x}) = \frac{A_3(\mathbf{x})}{A(\mathbf{x})}.$$

4.2 Properties

4.2.1 Interpolation

By definition of the shape function given in Eq. 30, the following property is selfevident:

$$0 \leq \Phi_I(\mathbf{x}) \leq 1. \tag{31}$$

Now, referring to Fig. 11, we note that if $\mathbf{x}$ were to coincide with any node, say node 3 for instance, then it is readily seen that $\Phi_3(\mathbf{x}) = 1$ and $\Phi_I(\mathbf{x}) = 0$, $I \neq 3$. Therefore, the shape functions share the following property:

$$\Phi_I(\mathbf{x}_J) = \delta_{IJ} \tag{32}$$

where δ_{IJ} is the *Kronecker symbol*.

4.2.2 Partition of Unity

By construction (Eq. 30), we have the following relation:

$$\sum_{I=1}^{\aleph} \Phi_I(\mathbf{x}) = 1 \text{ in } \Omega \tag{33}$$

where $\aleph$ is the number of natural neighbors for the point $\mathbf{x}$. Hence, by virtue of Eq. 31 and the above property, we note that the shape function form a partition of unity [10]. The implication is that this interpolant can exactly reproduce constant functions; in addition, the approximation space can also be enriched by additional functions.

4.3 Linear Completeness

For a second-order PDE such as elastoplastic, linear consistency or completeness is the ability of the interpolant to exactly reproduce constant and linear displacement fields. Sibson has shown that the natural neighbor shape functions satisfy the local coordinate property, namely

$$\mathbf{x} = \sum_{i=1}^{\aleph} \phi_i(\mathbf{x})\mathbf{x}_i \tag{34}$$

which indicates that the shape function can exactly reproduce the geometrical coordinates.

4.4 Supports and Natural Neighbors

Consider the point $I \in N$, where N is the set consisting of n natural neighbors for a point $\mathbf{x} \in \Omega \subset \Re^2$. The support or domain of influence of the shape function $\Phi_I(\mathbf{x})$ associated with the point I is defined as the closed sub-domain Ω_{sI} such that $\Phi_I(\mathbf{x}) > 0$ in Ω_{sI}, $\Phi_I(\mathbf{x}) = 0$ on $\partial\Omega_{sI}$, and $\Phi_I(\mathbf{x}) = 0$ in $CH(N) - \Omega_{sI}$. By the circumcircle criterion, it is evident that for $\Phi_I(\mathbf{x})$ to have a non zero contribution at $\mathbf{x}$, the point $\mathbf{x}$ must lie within the circumcircle of a Delaunay triangle that has node I as one of its vertices. It immediately follows from the above argument that the support of the shape function $\Phi_I(\mathbf{x})$ is the intersection of the convex hull $CH(N)$ with the union of all Delaunay circumcircles that pass through node I.

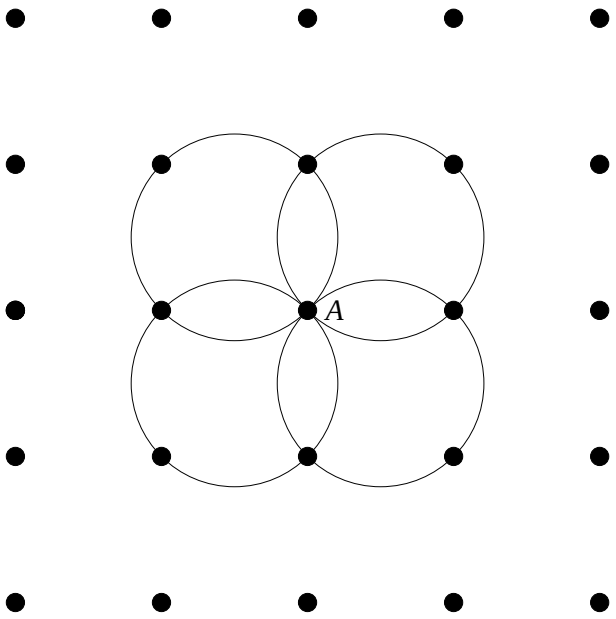

Figure 12: Support for the shape function.

In Fig. 12 a square is discretized by equi-spaced nodes. The support for the point A is the area of the circumcircles shown in the figure. Now in Fig. 13 we can see the shape function for node A which is located on the center, where $\phi_A(\mathbf{x}_A)$ takes on the value of unity. The support is clearly seen to be the union of the Delaunay circumcircles about node A. The surface of the shape function $\phi_A(\mathbf{x})$ is analogous to a taut rubber sheet that is stretched so as to meet the nodal data.

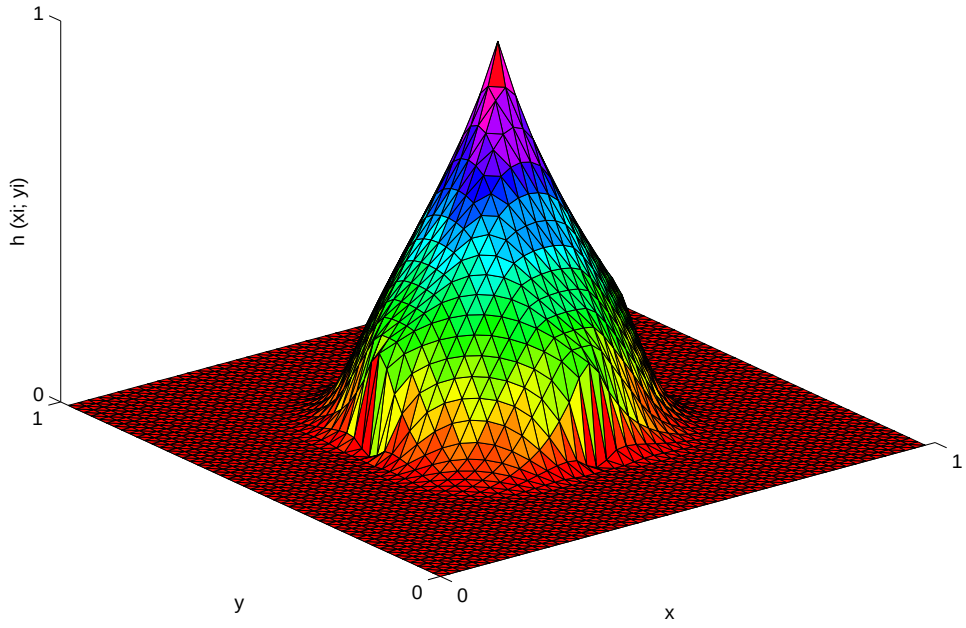

Figure 13: Shape function on the point A.

In Fig. 14 we can see two differents cuts of the shape function to appreciate its contour with more detail.

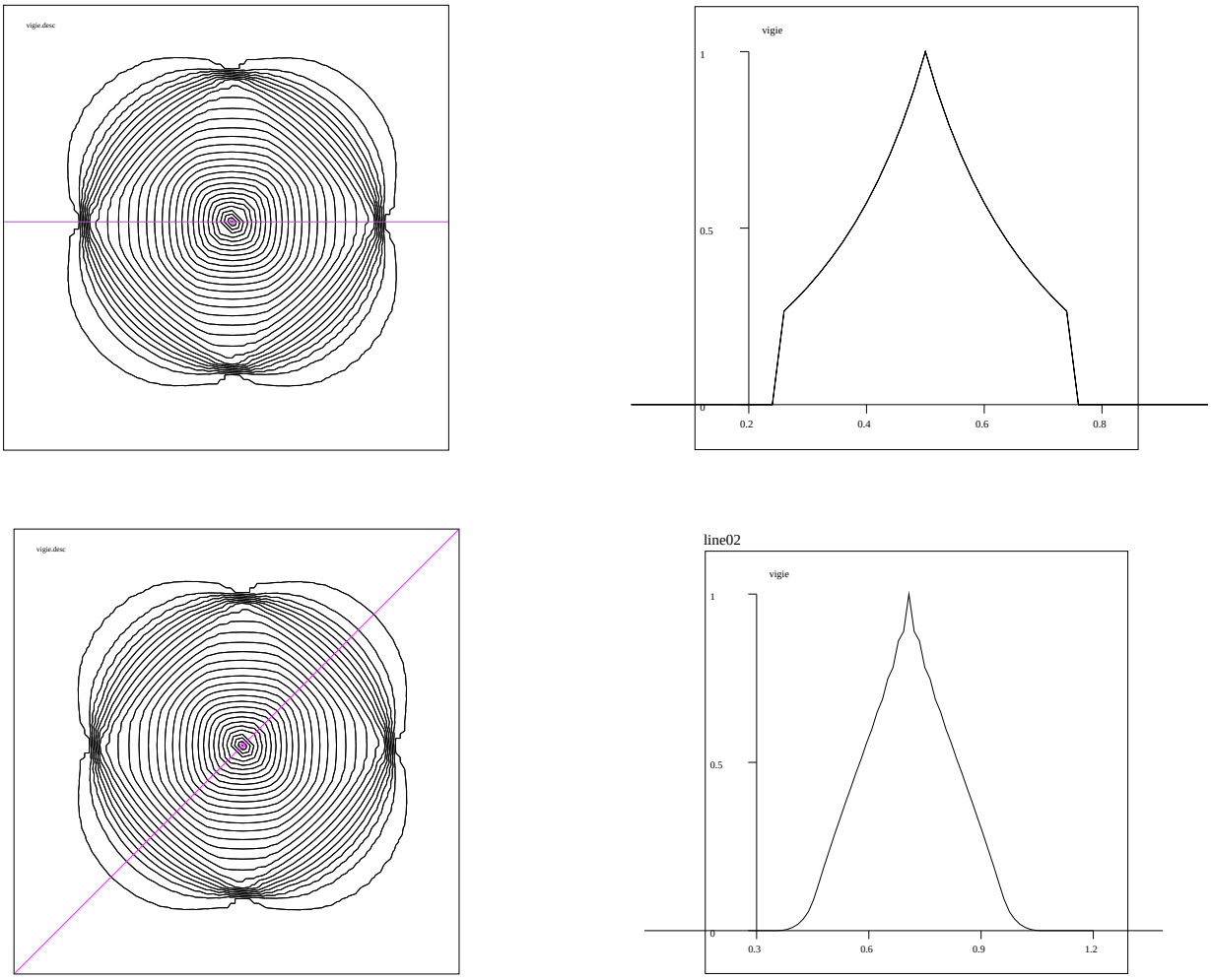

Figure 14: Detail of a transversal cut of the NNI shape function.

5 The Natural Neighbor Interpolation (NNI)

Assume that to each n_I of the set of points N is attached a continuously differentiable function h_I from $\Re^d$ to $\Re$. We define the NNI of the h_I [8] as

$$h(\mathbf{x}) = \sum_{I \in \aleph(\mathbf{x})} \Phi_I(\mathbf{x})\, h_I(\mathbf{x}) \tag{35}$$

where $\aleph$ are the natural neighbor of $\mathbf{x}$.

5.1 Why The Natural Neighbor Interpolation

Let us consider interpolations that are based on distance-based weights, such as the *Kernel Function* used in the MPS method. The weight function is isotropic (circular in 2D or spherical in 3D), non-negative within a circle or ball of some fixed radius, and monotonically decreasing with distance from the point $\mathbf{x}$. The rationale in this approach is that points that are closer to $\mathbf{x}$ are given a larger weight at $\mathbf{x}$ than those at farther distances.

Natural Neighbor Interpolation assumes a totally different viewpoint. The weight at a point $\mathbf{x}$ is not dictated by the same length measure in all dimensions, but by the appropriate Lebesgue measure of the space-dimension. This allows for anisotropic supports, where the size in direction $\mathbf{r}$ is not given by an L_2-metric but is ascertained based on a geometric construction that defines the region of interaction between the nodes. In MPS method (based on distance-based weights), the handling of irregular arrangements of nodes is non-trivial since contribution at a point $\mathbf{x}$ tends to be disproportionately biased towards areas of higher point density. In NNI, by virtue of construction, the distribution and density of points are taken into account by assigning weights to the points at a point $\mathbf{x}$. This concept is illustrated in Fig. 15 where B,C, and D are neighbors of A, but E and F are not, even though $d(A,E) < d(A,D) < d(A,C)$ and $d(A,F) < d(A,D) < d(A,C)$.

The number of natural neighbors n is a function of position $\mathbf{x}$ and also of the nodal density. In d-dimensions, the number of natural neighbors is at least $d+1$.

5.2 Smoothness

The smoothness or regularity of the shape function shown in Fig. 13 is discussed. Natural neighbor shape function are C^∞ everywhere, except at the nodes where they are C^0. Refering

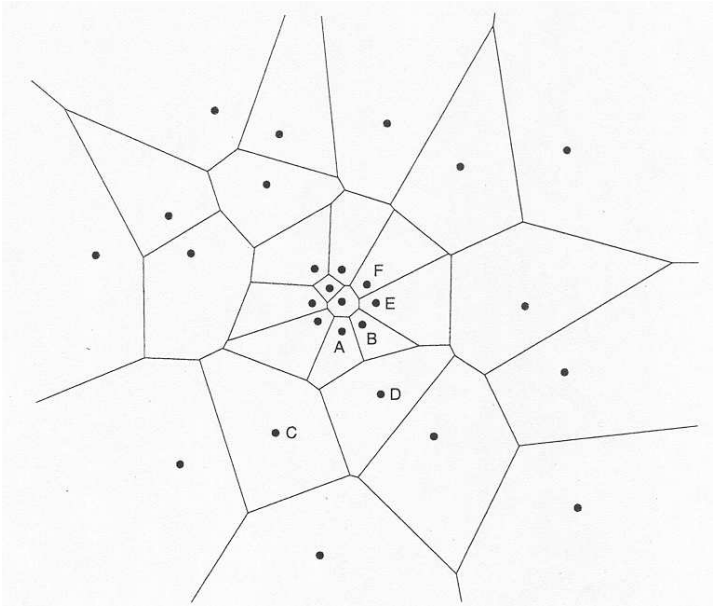

Figure 15: Voronoi neighbors.

to Fig. 11 and Fig. 14 we see that $\Phi_I(\mathbf{x})$ is a continuous function of $\mathbf{x}$. The only points of exception are the nodes, but since $\Phi_I(\mathbf{x}) = 1$, as $\mathbf{x} \to \mathbf{x}_I$ from any direction, the continuity of $\Phi_I(\mathbf{x})$ is established. The differentiability of the shape function $\Phi(\mathbf{x})$ at all points. But the nodes, is also evident. Since $\Phi(\mathbf{x})$ has compact support and is smoothly varying at all points except when approaching the nodes, its derivatives are C^∞ in Ω.

5.3 Interpolation in One-Dimension

In one-dimension, NNI as described earlier, is identical to linear finite elements. Let us prove this.

Proof. Consider a 1D bar of length L which is discretized by M unequally spaced nodes (Fig. 16). It is evident that the Voronoi vertices lay at the mid-point between any two adjacent nodes. In Fig. 16, the filled circles represent the nodes, while the bright circles are the Voronoi vertices. A consequence of the above observation is that all points in the open set (0,L) have two natural neighbors, while the points on the boundary have only one natural neighbor. In order to compute the shape functions, we consider the domain (element in FEM) Ω_I between any two adjacent nodes, say n_I and n_{I+1}. Let us use a normalized coordinate $\varepsilon = (x - x_I)/(x_{I+1} - x_I)$, where $\varepsilon \in [0, 1]$, and a local node numbering: $n_I \to 1$ and $n_{I+1} \to 2$ (Fig. 16). Consider a

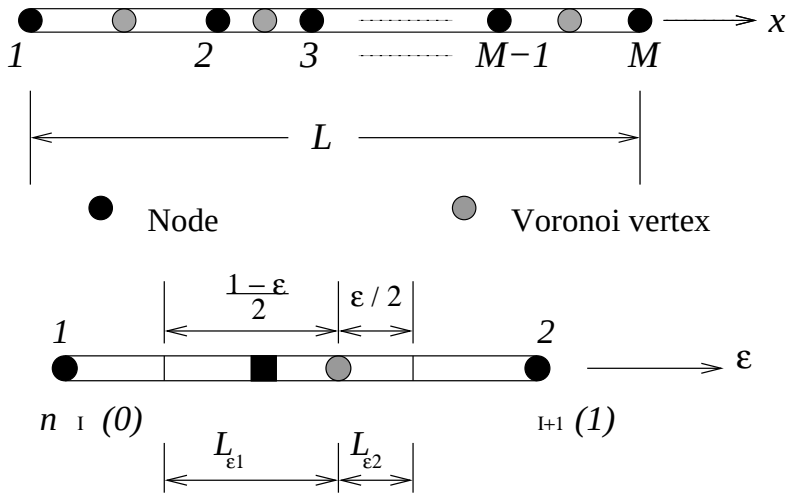

Figure 16: Shape functions in one-dimension for the physical space, and the reference space.

point $\varepsilon \in \Omega_I$. The second-order Voronoi cells about ε are shown in Fig. 16. Using Eq. 30, the shape function can be written as

$$\Phi(\varepsilon) = \frac{L_{\varepsilon I}}{L_{\varepsilon 1} + L_{\varepsilon 2}} \quad (I = 1, 2) \tag{36}$$

where $L_{\varepsilon 1} = (1 - \varepsilon)/2$ and $L_{\varepsilon 2} = \varepsilon/2$. On using these in the above equation, we obtain the result:

$$\Phi_1(\varepsilon) = 1 - \varepsilon\,, \quad \Phi_2(\varepsilon) = \varepsilon \tag{37}$$

which are precisely 1D linear finite element shape functions.

5.4 Interpolation in Two-Dimensions

The equivalence between the shape function shown in Fig. 13 to barycentric coordinates and bilinear interpolation, for the special cases of $n = 3$ and $n = 4$ (regular grid), respectively, are shown. For irregular quadrilaterals and $n > 4$, the shape functions are rational quadratic functions [12].

5.4.1 Three Natural Neighbors

If a point $\mathbf{x}$ has three natural neighbors ($n = 3$), then the shape functions are precisely the barycentric coordinates, or constant-strain triangle finite element shape functions.

Proof. By an argument of uniqueness for barycentric as well as NNI shape functions ($m = 3$), this equivalence is immediately seen[12]. Here we use the linear reproducing conditions to prove

the correspondence. Let the natural neighbors of point $\mathbf{x} = (x, y)$ be nodes 1, 2, and 3, with coordinates (x_I, y_I), $I = 1 - 3$ (Fig. 17). Using Eq. 33 and Eq. 34, the following conditions

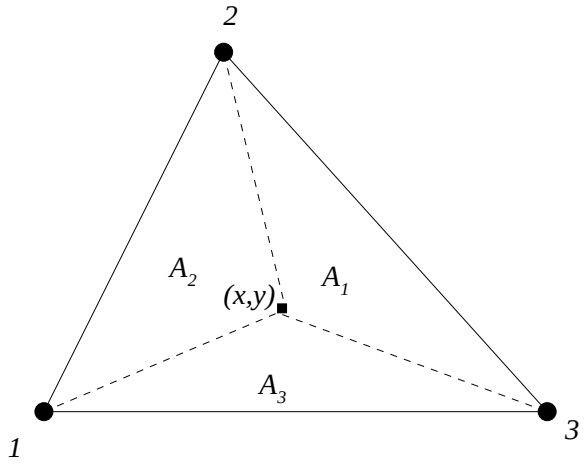

Figure 17: Barycentric coordinates ($n = 3$).

are met for the NNI shape function:

$$\sum_{I=1}^{3} \Phi_I(\mathbf{x}) = 1, \tag{38}$$

$$\sum_{I=1}^{3} \Phi_I(\mathbf{x}) x_I = x, \tag{39}$$

$$\sum_{I=1}^{3} \Phi_I(\mathbf{x}) y_I = y, \tag{40}$$

which in matrix form can be written as

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} \Phi_1(\mathbf{x}) \\ \Phi_2(\mathbf{x}) \\ \Phi_3(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix}$$

the solution of the above system of linear equations is:

$$\Phi_1(\mathbf{x}) = \frac{\mathbf{D}_1(\mathbf{x})}{\mathbf{D}(\mathbf{x})}, \tag{41}$$

$$\Phi_2(\mathbf{x}) = \frac{\mathbf{D}_2(\mathbf{x})}{\mathbf{D}(\mathbf{x})}, \tag{42}$$

$$\Phi_3(\mathbf{x}) = \frac{\mathbf{D}_3(\mathbf{x})}{\mathbf{D}(\mathbf{x})} \tag{43}$$

where

$$\mathbf{D}(\mathbf{x}) = \begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ x_1 & x_2 - x_1 & x_3 - x_1 \\ y_1 & y_2 - y_1 & y_3 - y_1 \end{vmatrix} = 2\mathbf{A}(\mathbf{x})$$

and $\mathbf{D}_1(\mathbf{x}) = 2\mathbf{A}_1(\mathbf{x})$, $\mathbf{D}_2(\mathbf{x}) = 2\mathbf{A}_2(\mathbf{x})$, and $\mathbf{D}_3(\mathbf{x}) = 2\mathbf{A}_3(\mathbf{x})$ (Fig. 17). $\mathbf{A}(\mathbf{x})$ is the area of $\triangle_{123}$. Hence we can write the shape function as

$$\Phi_1(\mathbf{x}) = \frac{\mathbf{A}_1(\mathbf{x})}{\mathbf{A}(\mathbf{x})}, \tag{44}$$

$$\Phi_2(\mathbf{x}) = \frac{\mathbf{A}_2(\mathbf{x})}{\mathbf{A}(\mathbf{x})}, \tag{45}$$

$$\Phi_3(\mathbf{x}) = \frac{\mathbf{A}_3(\mathbf{x})}{\mathbf{A}(\mathbf{x})} \tag{46}$$

which are precisely the barycentric coordinates for the point $\mathbf{x}$.

5.4.2 Four Natural Neighbors (Regular Grid)

For a regular rectangular nodal grid, if a point $\mathbf{x}$ has four natural neighbors ($n = 4$), then bilinear interpolation on the rectangle is obtained. A geometric proof is provided by Farin [12]. Here we use the definition of natural neighbor shape functions and explicitly carry out the computations to show the equivalence. It is to be noted that the above claim does not hold for the general case of four natural neighbors that are located at the vertices of a quadrilateral.

Proof. Consider a point $\mathbf{x}$ with four natural neighbors located at the vertices of a unit square: $(x_1, y_1) = (0, 0)$, $(x_2, y_2) = (1, 0)$, $(x_3, y_3) = (1, 1)$, $(x_4, y_4) = (0, 1)$ (Fig. 18). By definition of the NNI shape functions, we can write

$$\Phi_I(\mathbf{x}) = \frac{\mathbf{A}_I(\mathbf{x})}{\mathbf{A}(\mathbf{x})} \quad (I = 1 - 4), \tag{47}$$

where $\mathbf{A}_1(\mathbf{x})$, $\mathbf{A}_2(\mathbf{x})$, $\mathbf{A}_3(\mathbf{x})$, and $\mathbf{A}_4(\mathbf{x})$ are the areas of $\triangle_{eda}$, $\triangle_{eab}$, $\triangle_{ebc}$, and $\triangle_{ecd}$, respectively. In the above equation, $\mathbf{A}(\mathbf{x})$ is the area of the first-order Voronoi polygon *abcd* and e is the center of the unit square with coordinates $(1/2, 1/2)$.

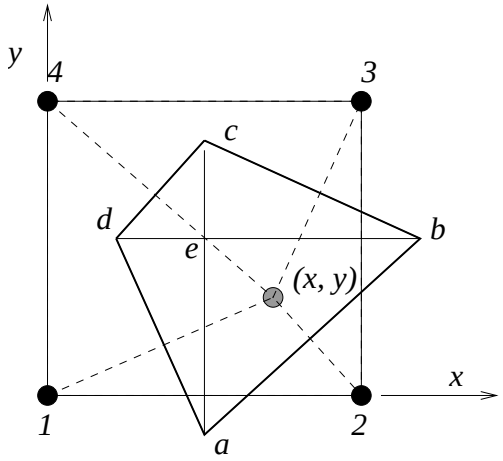

Figure 18: Bilinear interpolation on a regular grid ($n = 4$).

We proceed by computing the area of the second-order Voronoi cells. To this end, by recalling the construction of $2nd$-order Voronoi cells, it is clearly seen that vertex a is the center of the circle that circumscribes the triangle $\triangle_{12\mathbf{x}}$, and proceeding likewise, b is the circumcenter of the triangle $\triangle_{23\mathbf{x}}$, c that of $\triangle_{34\mathbf{x}}$, and d that of $\triangle_{41\mathbf{x}}$. To compute the circumcenter of a triangle $\triangle_{\alpha\beta\gamma}$ lets use the expressions

$$v_1 = \frac{(\alpha_1^2 - \gamma_1^2 + \alpha_2^2 - \gamma_2^2)(\beta_2 - \gamma_2) - (\beta_1^2 - \gamma_1^2 + \beta_2^2 - \gamma_2^2)(\alpha_2 - \gamma_2)}{[(\alpha_1 - \gamma_1)(\beta_2 - \gamma_2) - (\beta_1 - \gamma_1)(\alpha_2 - \gamma_2)]} \tag{48}$$

$$v_2 = \frac{\beta_1^2 - \gamma_1^2 + \beta_2^2 - \gamma_2^2)(\alpha_2 - \gamma_2) - (\alpha_1^2 - \gamma_1^2 + \alpha_2^2 - \gamma_2^2)(\beta_1 - \gamma_1)}{[(\alpha_1 - \gamma_1)(\beta_2 - \gamma_2) - (\beta_1 - \gamma_1)(\alpha_2 - \gamma_2)]} \tag{49}$$

so this coordinates are computed to be

$$a_1 = 1/2, \quad a_2 = \tfrac{-x+x^2+y^2}{2y} \tag{50}$$

$$b_1 = \frac{1 + y - x^2 - y^2}{2(1-x)}, \qquad b_2 = 1/2 \tag{51}$$

$$c_1 = 1/2, \quad c_2 = \tfrac{1+x-x^2-y^2}{2(1-y)} \tag{52}$$

$$d_1 = \frac{-y + x^2 + y^2}{2x}, \qquad d_2 = 1/2 \tag{53}$$

Given the formula for the area of a triangle $\triangle_{\alpha\beta\gamma}$ like

$$\mathbf{A} = \frac{(\alpha_1 - \gamma_1)(\beta_2 - \gamma_2) - (\beta_1 - \gamma_1)(\alpha_2 - \gamma_2)}{2} \tag{54}$$

we obtain

$$\mathbf{A}_1(\mathbf{x}) = \frac{(x^2+y^2-x-y)^2}{8xy}, \tag{55}$$

$$\mathbf{A}_2(\mathbf{x}) = \frac{(x^2+y^2-x-y)^2}{8y(1-x)}, \tag{56}$$

$$\mathbf{A}_3(\mathbf{x}) = \frac{(x^2+y^2-x-y)^2}{8(1-x)(1-y)}, \tag{57}$$

$$\mathbf{A}_4(\mathbf{x}) = \frac{(x^2+y^2-x-y)^2}{8x(1-y)}. \tag{58}$$

Since $\mathbf{A}(\mathbf{x}) = \sum_{I=1}^{4} \mathbf{A}_I(\mathbf{x})$, we have

$$\mathbf{A}(\mathbf{x}) = \frac{(x^2+y^2-x-y)^2}{8xy(1-x)(1-y)} \tag{59}$$

and hence using Eq. 58 and Eq. 59 in Eq. 47, the NNI shape function can be written as

$$\Phi_1(\mathbf{x}) = (1-x)(1-y), \tag{60}$$

$$\Phi_2(\mathbf{x}) = x(1-y), \tag{61}$$

$$\Phi_3(\mathbf{x}) = xy, \tag{62}$$

$$\Phi_4(\mathbf{x}) = y(1-x), \tag{63}$$

which are precisely bilinear FE shape functions. The above derivation is easily generalized to the rectangle (linear transformation of a square), and hence bilinear interpolation on the rectangle is realized by the Natural Neighbor Interpolation.

6 Numerical Computation Procedure for the NNI Shape Function

As it was mentioned before, the C^{++} *CGAL* library was used to compute the interpolation. The *CGAL* library contains a number of different parts.

1. The elementary part of the library (the kernel) consist of primitive, constant-size geometric objects (points, lines, spheres, etc.) and predicates on them (orientation test for points, intersection tests, etc.).

2. The subsequent part of the library contains a number of standard geometrical algorithms and data structures such as convex hull, smallest enclosing circle and triangulation.

3. The last part of the library consists of a support library for example for I/O, visualization and random generators.

Currently the library contains mainly 2 and 3-dimensional objects.

Some characteristics of this library are:

1. Robustness. A correct result of an algorithm can be guaranteed if geometric predicatives are evaluated exactly (managing the round-off errors in a correct way). To do this, *CGAL* chose an appropriate number type for doing computations. It is always clear for which inputs and for which number types a correct result is guaranteed.

2. Generality. The user choses the appropriate number type for doing computations. This will depends on which are the priorities, for example if speed is important, computation can be done with floats or doubles.

3. Efficiency. Multiple versions of an algorithm are supplied.

Farther detail are given in the *CGAL* documentation [6].

Now the steps for the interpolation will be described.

6.1 Defining the Triangulation

Let us consider a set of points $N = n_1, n_2, \ldots, n_m$ in $\Re^2$. The basic triangulation class of *CGAL* is primarily designed to represent the triangulations of the set of points N in the plane.

Such a triangulation has as its vertices the points of N and its domain covers the convex hull of N. It can be viewed as a planar partition of the plane whose bounded faces are triangular and cover the convex hull of N. The single unbounded face of this partition is the complementary of the convex hull of N. A vertex called infinite vertex is added to the triangulation a fictitious vertex, and each convex hull edge aims to an infinite face having as third vertex the infinite vertex. In that way, each edge is incident to exactly two faces and special cases at the boundary of the convex hull are simpler to deal with (Fig. 19).

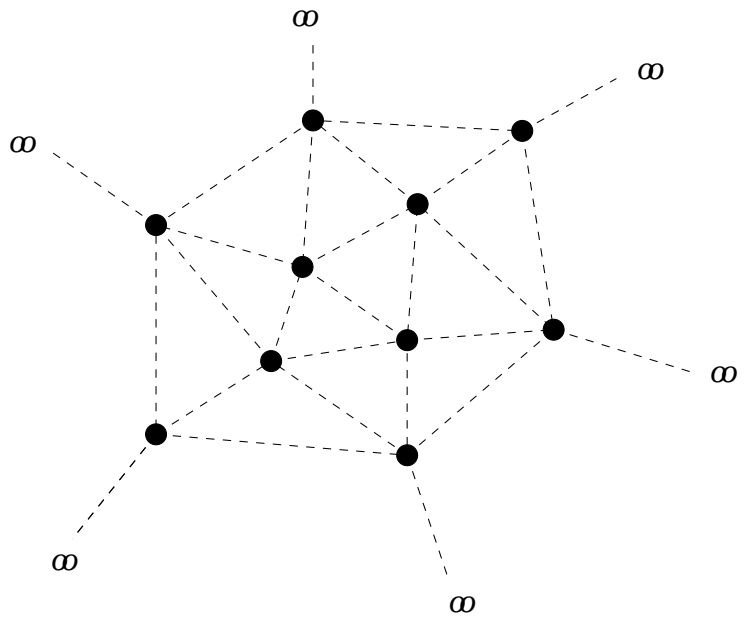

Figure 19: Infinite Vertices.

A triangulation is a collection of vertices and faces that are linked together through incidence and adjacency relations. Each face give access to its three incident vertices and to its three adjacent faces. Each vertex give access to one of its incident faces.

Now that we have defined the basic triangulation in $CGAL$, we want to insert the points in a given triangulation declared as tr. So each point of the set of points N is going to be inserted by $tr.insert(n_i)$. Once all the points have been inserted in the triangulation, we may want to access the vertices of the triangulation. For example, if we need the coordinates, the method $v.point()$ gives the point associated with the vertex v. All the finite vertices of the triangulation can be accessed through a vertex iterator, defined within the triangulation. So if an iterator it is defined, its value type is going to be a vertex of the triangulation. The method

vertices_begin() gives an iterator referring to the first vertex in the range according to the triangulation data structure. *vertices_end*() gives the past-the-end iterator of the range.

It is possible to declare iterators for the faces and the edges of the triangulation in an analog way.

6.1.1 The Delaunay Triangulation

The successive insertion of points in the triangulation can result in very elongated triangles. To avoid this, we want the minimum angle over all the triangles not to be too small. The Delaunay triangulation maximizes the minimum angle.

The Delaunay triangulation inherits from the basic triangulation structure defined earlier. It redefines the insert and remove functions. When a new point is inserted, the face containing the point is not just split, but the triangulation is modified as to satisfy the max-min angle property.

6.1.2 Convex Hull

The convex hull is the set of objects of the set of points N that is the smallest convex object that contains all objects of the set. In *CGAL* the convex hull is defined by the vertices that have as neighbor the infinite vertex (Fig. 20).

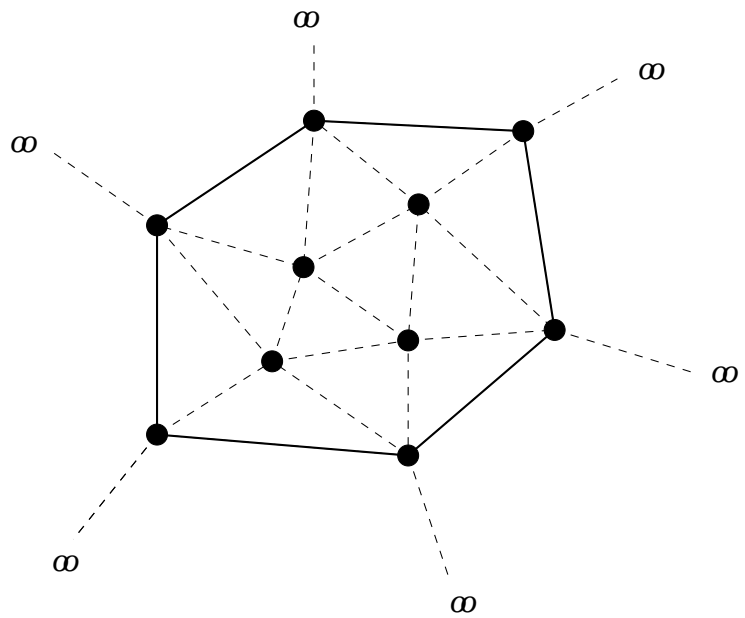

Figure 20: Convex Hull.

6.2 Computing the Interpolation

According to the steps that we have followed above, we have a triangulation represented by tr where all the points of the set N have been inserted. The next step is to compute the interpolation. To this end, the function $Natural\_neighbors\_2$ will be used. This function interpolates a given point n_j that does not belong to the triangulation, with the others point of the triangulation.

In a problem where the domain Ω has been discretized in a finite number of points, the interpolation consists in relating each point that belong to Ω with the other points that belong to the discretization. To do this we use the iterators to go around all the points of the triangulation and evaluate the interpolation with its natural neighbors. This is not possible in a direct way by applying the $Natural\_neighbors\_2$ function. This is logical because if we define an iterator it, this iterator represents a vertex in the triangulation and when the interpolation is evaluated and we give this point to the function $Natural\_neighbors\_2$, the result will be

$$h(\mathbf{x}) = \sum_{i \in \aleph(\mathbf{x})} \Phi_I(\mathbf{x}) = 1 \tag{64}$$

where $\aleph$ are the natural neighbor of $\mathbf{x}$ and $\aleph = 1$. In other words, the point associated with it interpolates with itself and by Eq. 33 the interpolation is equal to the unity (Fig. 13).

To solve this problem a new *local* triangulation is constructed where only the points that are the incident vertices to it in the triangulation are inserted. So we are introducing a new concept in the $CGAL$ library, the circulators. Given a vertex v it is possible to declare a circulator $circ$ that is going to circulate through the vertices of the triangulation incident to v. Fig. 21 provides an example where i will be interpolated with its neighbors j. So from the triangulation tr another triangulation called local triangulation is built and then the interpolation takes part according to the construction described in Fig. 11.

So the steps for the interpolation are summaryed as follows:

1. Declare the variable which will store the triangulation (Ex. tr);
2. insert the points of the domain Ω in the triangulation;
3. iterate through all the vertices in the triangulation;
4. circulate around each vertex to build the local triangulation;

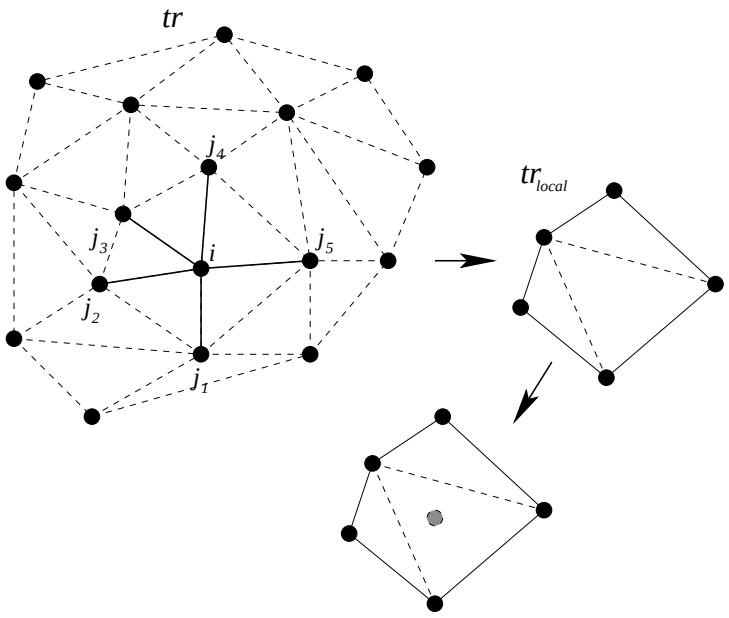

Figure 21: Local triangulation to solve the Interpolation.

5. compute the interpolation.

6.3 Programming in C^{++}

As *CGAL* is a library programmed in the C^{++} language the code to compute the interpolation had to be done in C^{++} too. Lots of advantages were found by doing this.

Due to the object orientated concept introduced by C^{++} an object can have different unique characteristics that differentiate it from the rest. For example the vertices in the triangulation are objects and each of these vertices represents a discret element of the continuous media. That means that the vertices have physical characteristics like pressure, density, velocity, temperature, etc. For example in numerical methods it is essential to assign an index or label to each vertex for the assembling of the system of equations. This makes the programming a lot more intuitive than to work with vectors and matrices and also simplify the debugging, as one can assign a new characteristic to each object at any time and in that way compare results that may have no physical meaning but a numerical meaning. To do this the class *Triangulation_vertex_base_2* was modified.

7 Particle Method Based on the Natural Neighbor Interpolation Shape Function

In section 2.2 an interpolation was defined to approximate the Navier Stokes equations. The laplacian of pressure and the gradients of pressure were approximated according to a function called kernel function (Eq. 3).

Then in section 5.1 we gave some reasons of why another kind of interpolation such as NNI could be more appropriate to model the laplacian and the gradients. The NNI was explained in detail. Now the NNI is used in the approximation of linear elliptic problems.

Taking Eq. 8 and Eq. 13 seen in section 2 the interpolation given by Eq. 3 is changed by using the NNI:

$$h(\mathbf{x}) = \sum_{I \in \aleph(\mathbf{x})} \Phi_I(\mathbf{x})\, h_I(\mathbf{x}) \tag{65}$$

So, the discrete approximation of the gradient was given by:

$$\langle \nabla \phi \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \Big[\frac{\phi_j - \phi_i}{|\mathbf{r}_j - \mathbf{r}_i|^2} \, (\mathbf{r}_j - \mathbf{r}_i)\, w(|\mathbf{r}_j - \mathbf{r}_i|) \Big] \tag{66}$$

where the summation involved all the particles in the vicinity of i (Fig. 2). This concept is now changed and the natural neighbor concept considered. So changing the function $w(|\mathbf{r}_j - \mathbf{r}_i|)$ by the natural neighors shape function $\Phi_I(\mathbf{x})$ we obtain the next expression for the gradient

$$\langle \nabla \phi \rangle_i = \frac{d}{\eta} \sum_{j=1}^{\aleph} \Big[\frac{\phi_j - \phi_i}{|\mathbf{r}_j - \mathbf{r}_i|^2} \, (\mathbf{r}_j - \mathbf{r}_i)\, \Phi_j(\mathbf{x}_i) \Big]. \tag{67}$$

where $h_{Ij}(\mathbf{x})$ was replaced by $d/n^0 \frac{(\phi_j - \phi_i)(\mathbf{r}_j - \mathbf{r}_i)}{|\mathbf{r}_j - \mathbf{r}_i|^2}$.

Proceeding in the same way for the laplacian and changing the interpolation function

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{\eta \lambda} \sum_{j=1}^{\aleph} (\phi_j - \phi_i)\, \Phi_j(\mathbf{x}_i), \tag{68}$$

where d is the number of space dimensions (2 in 2D), $\aleph$ is the number of natural neighbors, $\eta = \sum_{j=1}^{\aleph} \Phi_j(\mathbf{x}_i)$. In this case the value of $h_{Ij}(\mathbf{x}) = \frac{2d}{\eta \lambda}(\phi_j - \phi_I)$.

Following the property of partition of unity (Eq. 33), we express η as

$$\eta = \sum_{j=1}^{\aleph} \Phi_j(\mathbf{x}_i) = 1$$

Thus, according to Eq. 17, λ can be expressed by

$$\lambda = \sum_{j=1}^{\aleph} |\mathbf{r}_j - \mathbf{r}_i|^2 \, \Phi_j(\mathbf{x}_i)$$

The final expression for the gradient and the laplacian by the NNI will be:

$$\langle \nabla^2 \phi \rangle_i = \frac{2d \sum_{j=1}^{\aleph} (\phi_j - \phi_i) \, \Phi_j(\mathbf{x}_i)}{\sum_{j=1}^{\aleph} |\mathbf{r}_j - \mathbf{r}_i|^2 \, \Phi_j(\mathbf{x}_i)} \tag{69}$$

$$\langle \nabla \phi \rangle_i = d \sum_{j=1}^{\aleph} \Big[\frac{\phi_j - \phi_i}{|\mathbf{r}_j - \mathbf{r}_i|^2} \, (\mathbf{r}_j - \mathbf{r}_i) \, \Phi_j(\mathbf{x}_i) \Big] \tag{70}$$

7.1 Order of Convergence for the NNI Approximation of Laplacian

In order to illustrate the analysis, let us consider the Poisson equation with Dirichlet boundary conditions:

$$-\nabla^2\phi = f \; in \; \Omega \tag{71}$$

$$\phi = 0 \; on \; \Gamma \tag{72}$$

where $\Gamma \subset \Re$ is an open bounded domain and the data f is assumed to be sufficiently smooth.

We are going to show that the NNI approximation of laplacian (Eq. 70) has the same order of approximation of a finite difference five nodes scheme in a 2D squared and equi-spaced grid (Fig. 22).

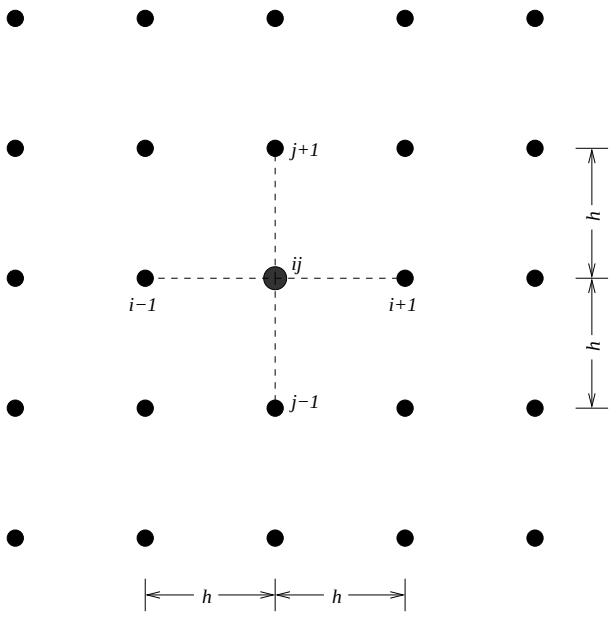

Figure 22: Finite Difference 5 nodes scheme.

To do this we will find the order of convergence for the finite difference scheme and then we will show that in the conditions explained above the NNI approximation has exactly the same expression as the finite difference scheme, thus the some order of the approximation.

7.1.1 Finite Difference Order of the Convergence for the Poisson Equation

To model Eq. 72 it is necessary to find an expression for the second order derivatives of ϕ. We are doing this by expanding ϕ in a Taylor series and following the scheme shown in Fig. 22. Let us expand ϕ around the point i, such that

$$\phi_{i-1\,j} \approx \phi_{i\,j} - \phi'_{i\,j}h + \frac{\phi''_{i\,j}h^2}{2} - \frac{\phi'''_{i\,j}h^3}{3!} + \frac{\phi^{IV}_{i\,j}h^4}{4!} - \cdots \quad (73)$$

$$\phi_{i+1\,j} \approx \phi_{i\,j} + \phi'_{i\,j}h + \frac{\phi''_{i\,j}h^2}{2} + \frac{\phi'''_{i\,j}h^3}{3!} + \frac{\phi^{IV}_{i\,j}h^4}{4!} + \cdots \quad (74)$$

adding both expressions we get

$$\phi_{i-1\,j} + \phi_{i+1\,j} \approx 2\phi_{i\,j} + 2\frac{\phi''_{i\,j}h^2}{2} + 2\frac{\phi^{IV}_{i\,j}h^4}{4!} + \cdots \quad (75)$$

giving the approximation for the second derivative of ϕ with respect to x

$$\frac{\partial^2\phi}{\partial x^2} \approx \frac{\phi_{i-1\,j} - 2\phi_{ij} + \phi_{i+1\,j}}{h^2} + O(h^2) \quad (76)$$

By doing the same operations in the y direction we get an anologous expression for the derivative of ϕ with respect to y

$$\frac{\partial^2\phi}{\partial y^2} \approx \frac{\phi_{i\,j-1} - 2\phi_{ij} + \phi_{i\,j+1}}{h^2} + O(h^2) \quad (77)$$

To get the approximation of the Poisson equation (Eq. 72) we add Eq. 76 and Eq. 77 to get

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} \approx \frac{\phi_{i-1\,j} + \phi_{i+1\,j} - 4\phi_{ij} + \phi_{i\,j-1} + \phi_{i\,j+1}}{h^2} + O(h^2). \quad (78)$$

7.1.2 NNI in a Rectangular Equi-Spaced Grid

The special and very simple case of a rectangular does not represent a challenge for the NNI approximation of the Eq. 70. Although we are going to do that in order to show that in this situation the NNI scheme is reduced to the finite difference scheme and therefore we can compare the order of approximations. Let us write again Eq. 70 and analyze each of its members:

$$\langle\nabla^2\phi\rangle_i = \frac{2d\sum_{j=1}^{\aleph}(\phi_j - \phi_i)\,\Phi_j(\mathbf{x}_i)}{\sum_{j=1}^{\aleph}|\mathbf{r}_j - \mathbf{r}_i|^2\,\Phi_j(\mathbf{x}_i)} \quad (79)$$

In Fig. 23 a general case for an equi-spaced grid is shown. Now if we insert a point i as shows Fig. 24 and we construct the second order Voronoi diagram we can interpolate the point with its 8 neighbors. By definition of the natural coordinates

$$\Phi_j(\mathbf{x}_i) = \frac{A_j}{A(\mathbf{x}_i)}$$

where j are the natural neighbors of i and $A(\mathbf{x}_i) = \sum_{j=1}^{8} A(\mathbf{x}_j)$ for the case in Fig. 24.

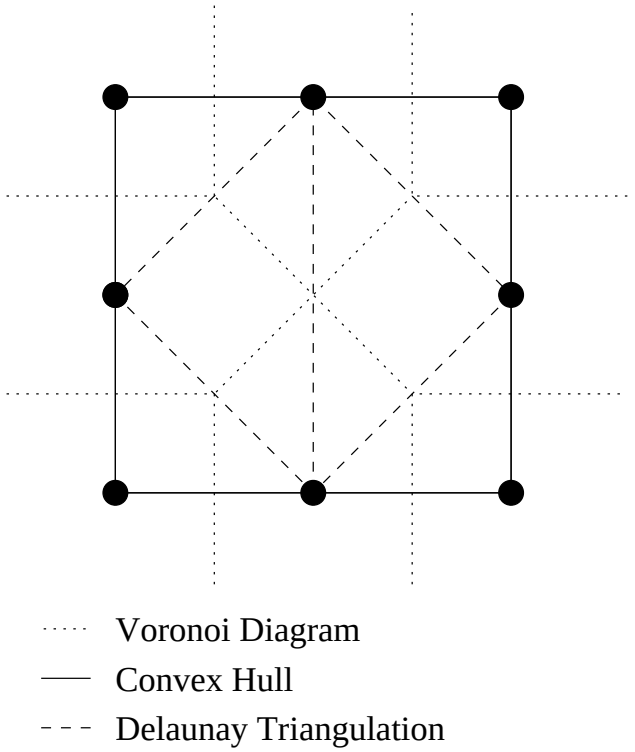

Figure 23: Local triangulation for 8 equi-spaced points.

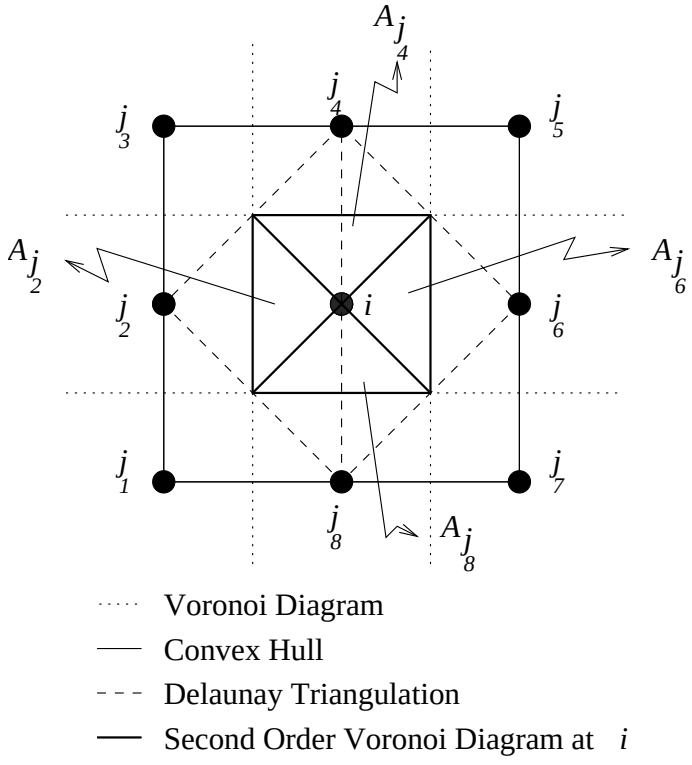

Figure 24: Interpolation of i with its $j-th$ neighbors.

then as the points $A_{j1} = A_{j3} = A_{j5} = A_{j7} = 0$. Due to the fact of construction, for an equi-sapaced grid it is easy to see that

$$A_{j2} = A_{j4} = A_{j6} = A_{j8} = A_j$$

and thus

$$A(\mathbf{x}_i) = A_{j2} + A_{j4} + A_{j6} + A_{j8} = 4A_j$$

and then the natural coordinates for each point will be

$$\Phi_j(\mathbf{x}_i) = \frac{A_j}{A(\mathbf{x}_i)} = \frac{A_j}{4A_j} = \frac{1}{4}. \tag{80}$$

now that we know the values of the Φ_js for all the neighbors, let us see the other term in equation Eq. 70. In Cartesian coordinates, one can evaluate $|\mathbf{r}_j - \mathbf{r}_i|^2$ like

$$d^2 = (x_j^2 - x_i^2) + (y_j^2 - y_i^2)$$

where d is the distance between i and j. In an equi-spaced rectangular grid we have that

$$x_i = x_j \pm h, \quad y_i = y_j \pm h \tag{81}$$

where h is shown in Fig. 25

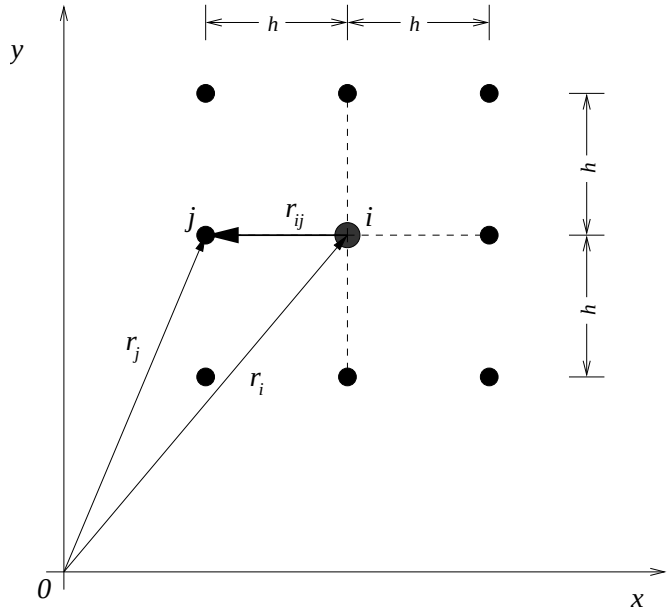

Figure 25: Distance between points to evaluate Eq. 70.

thus,

$$d^2 = |\mathbf{r}_j - \mathbf{r}_i|^2 = h^2$$

We are ready now to expand the summation in Eq. 70 for the 8 natural neighbors of i, get the expression for laplacian:

$$\langle \nabla^2 \phi \rangle_i = \frac{4 \sum_{j=1}^{8} (\phi_j - \phi_i)\, \Phi_j(\mathbf{x}_i)}{\sum_{j=1}^{8} h^2 \Phi_j(\mathbf{x}_i)} \tag{82}$$

$$= \frac{4((\phi_{j2} - \phi_i)0.25 + (\phi_{j4} - \phi_i)0.25 + (\phi_{j6} - \phi_i)0.25 + (\phi_{j8} - \phi_i)0.25)}{4(0.25h^2)} \tag{83}$$

where the terms having zero contribution were not written. By distributing and regruping terms the next expression is obtained

$$\langle \nabla^2 \phi \rangle_i = \frac{4}{h^2}(0.25\phi_{j2} + 0.25\phi_{j4} + 0.25\phi_{j6} + 0.25\phi_{j8} - \phi_i) \tag{84}$$

the values of ϕ_{j2}, ϕ_{j4}, ϕ_{j6}, and ϕ_{j8} correspond to the values of $\phi_{i-1\,j}$, $\phi_{i\,j+1}$, $\phi_{i+1\,j}$, and $\phi_{i\,j-1}$ respectively according to Fig. 24 and Fig. 22.

Eq. 78 representing the finite difference scheme and Eq. 84 representing the NNI scheme for a rectangular equi-spaced grid are the same and if Eq. 78 had an order of approximation of $O(h^2)$ then Eq. 84 has the same order of approximation.

So we have shown that the NNI in a rectangular grid equi-spaced has a second order of convergence and therefore it converges in this particular case. Later, more complicated domains will be analyzed and the results will be compared to the FEM which has a very solid analysis of convergence in such domains.

7.2 Order of Convergence for the NNI Approximation of the Gradient

Following the same logic that we have used to find the order of the convergence of the NNI for the expression of laplacian we are now going to find the order of the convergence for the gradient. The analysis will be done over the same grid shown in Fig. 22. So first the analysis will be carried out for the finite difference method and then we will try to arrive to a similar expression using the gradient model for NNI.

7.2.1 Finite Difference Order of Convergence for the Gradient

Expanding the Taylor series of ϕ around i we will find an order of approximation for a centered finite difference scheme. So in a similar way to what we have done above for the laplacian we have

$$\phi_{i+1\,j} \approx \phi_{i\,j} + \phi'_{i\,j}h + \frac{\phi''_{i\,j}h^2}{2} + \frac{\phi'''_{i\,j}h^3}{3!} + \cdots \tag{85}$$

$$\phi_{i-1\,j} \approx \phi_{i\,j} - \phi'_{i\,j}h + \frac{\phi''_{i\,j}h^2}{2} - \frac{\phi'''_{i\,j}h^3}{3!} + \cdots \tag{86}$$

subtracting both expressions

$$\phi_{i+1\,j} - \phi_{i-1\,j} = 2\phi'_{i\,j}h + 2\frac{\phi'''_{i\,j}h^3}{3!} + \cdots \tag{87}$$

then the first derivative with respect to x will be written as

$$\frac{\partial\phi}{\partial x} \approx \frac{\phi_{i+1\,j} - \phi_{i-1\,j}}{2h} + O(h^2) \tag{88}$$

and the order of convergence for this centered scheme will be $O(h^2)$.

The analogous procedure has to be done for the derivative of $\frac{\partial\phi}{\partial y}$ and the expression

$$\frac{\partial\phi}{\partial y} \approx \frac{\phi_{i\,j+1} - \phi_{i\,j-1}}{2h} + O(h^2) \tag{89}$$

is found.

7.3 Convergence for the NNI Gradient

Let us remember the expression for the modeling of gradient

$$\langle\nabla\phi\rangle_i = d\sum_{j=1}^{\aleph}\Big[\frac{\phi_j - \phi_i}{|\mathbf{r}_j - \mathbf{r}_i|^2}\,(\mathbf{r}_j - \mathbf{r}_i)\,\Phi_j(\mathbf{x}_i)\Big] \tag{90}$$

where for this case $d = 2$ (2D problem) and it was shown above that

$$|\mathbf{r}_j - \mathbf{r}_i|^2 = h^2$$

It was also proved that

$$\Phi_j(\mathbf{x}_i) = \frac{A_j}{A(\mathbf{x}_i)} = \frac{A_j}{4A_j} = \frac{1}{4}. \tag{91}$$

The values for the vector $\mathbf{r}_j - \mathbf{r}_i$ can be deduced from Fig. 25 in a very simple way by the use of the cartesian expressions Eq. 81 and we get the values

$$\begin{aligned}
\mathbf{r}_{j2} - \mathbf{r}_i &= (-h, 0), \\
\mathbf{r}_{j4} - \mathbf{r}_i &= (0, h), \\
\mathbf{r}_{j6} - \mathbf{r}_i &= (h, 0), \\
\mathbf{r}_{j8} - \mathbf{r}_i &= (0, -h).
\end{aligned}$$

Now it is possible to write the gradient with all its terms for the case of the rectangular grid without writing the terms with null contribution to the interpolation as we have done for the laplacian

$$\begin{aligned}
\langle \nabla\phi \rangle_i \; = 2 \sum_{j=1}^{8} \left[\frac{\phi_j - \phi_i}{h^2} (\mathbf{r}_j - \mathbf{r}_i)\, 0.25 \right] = 2 \Big(& \frac{(\phi_{j2} - \phi_i)(-h) 0.25}{h^2} + \frac{(\phi_{j6} - \phi_i)(h) 0.25}{h^2}; \\
& \frac{(\phi_{j4} - \phi_i)(h) 0.25}{h^2} + \frac{(\phi_{j8} - \phi_i)(-h) 0.25}{h^2} \Big)
\end{aligned} \tag{92}$$

Splitting the gradient into its two components we get

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{j6} - \phi_{j2}}{2h} \tag{93}$$

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{j4} - \phi_{j8}}{2h}. \tag{94}$$

Like in the case of the laplacian, the values of ϕ_{j2}, ϕ_{j4}, ϕ_{j6}, andϕ_{j8} correspond to the values of $\phi_{i-1\,j}$, $\phi_{i\,j+1}$, $\phi_{i+1\,j}$, and$\phi_{i\,j-1}$ respectively according to Fig. 24 and Fig. 22.

Eq. 88 and Eq. 89 representing the finite difference scheme and Eq. 94 and Eq. 94 representing the NNI scheme for the gradient of ϕ in a rectangular equi-spaced grid are the

same and if Eq. 88 and Eq. 89 had an order of approximation of $O(h^2)$ then Eq. 94 and Eq. 94 have the same order of convergence.

We have shown that both, the laplacian and the gradient of a scalar quantity ϕ modeled by the NNI have a second order of convergence for a rectangular equi-spaced grid.

In the next sections we exame some numerical results and comparisons of different solutions for different problems solved by differents method. A discussion about the convergence and quality of the results is provided..

8 Numerical Results and Applications

Two basic problems were carried out in order to test the efficiency of the method and to compare the results to methods like FEM, where the results are well known, and for cases where we know that FEM converges giving good approximate results.

Several cases are studied in basically two different problems:

1. The first one is a square domain (Fig. 26) where the Poisson equation is solved. A simple domain is chosen so the solution can also be compared with a traditional finite difference approximation like the one deduced above, where we proved that the method has a second order of convergence. For this case, the solutions obtained by the finite element, finite difference and natural neighbor interpolation can be compared. Then for the same problem the distribution of points is going to be changed to refine the grid where the changes of gradients take place and the finite element and natural neighbor interpolation are compared in this domain.

2. The second example is a nozzle geometry Fig. 27 which is a more complicated geometry. Both finite element and natural neighbors interpolation will be compared for different refinements.

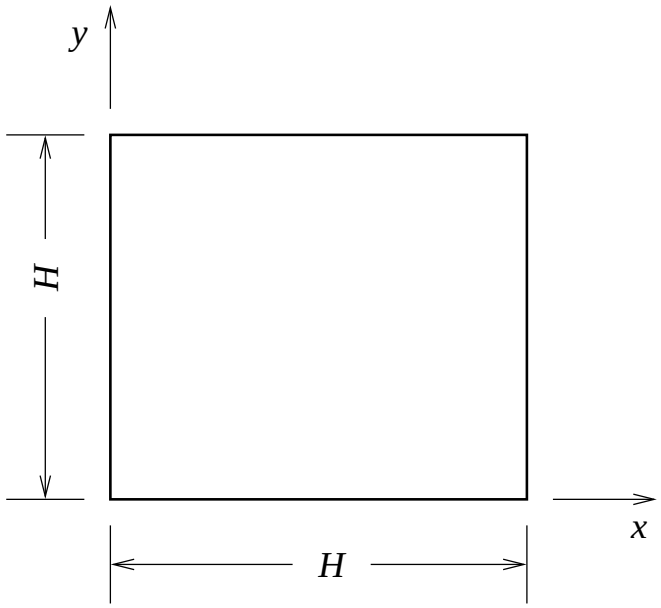

Figure 26: Geometry for the square domain

For both cases the numerical results are compared.

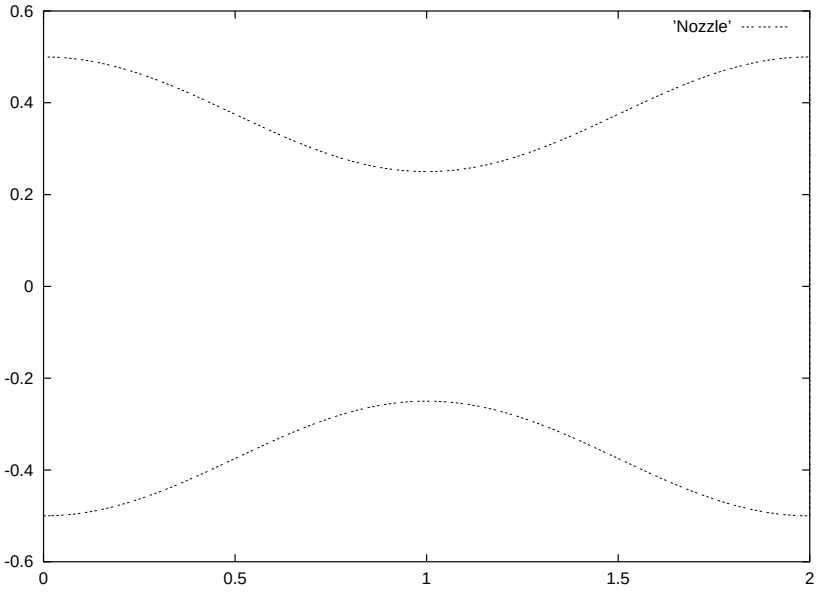

Figure 27: Nozzle geometry.

8.1 Approximation of Poisson Equation over an Square Domain

We consider the two dimensional Poisson equation with Dirichlet boundary conditions

$$\nabla^2\phi = f(x,y) \text{ if } \phi \text{ in } \Omega \tag{95}$$

$$\phi = g(x,y) \text{ if } \phi \text{ on } \Gamma \tag{96}$$

As the example that we introduce here is only to compare solutions, no special physical meaning was assigned to the $g(x,y)$ function or to the boundary conditions. We tried to solve a representative case where the priority was the comparison of results. The following choices were made:

$$f(x,y) = \begin{cases} 1 \quad if \quad (3/8\,H \leqq x \leqq 5/8\,H,\ \ 3/8\,H \leqq y \leqq 5/8\,H) \\ \\ 0 \quad if \quad (3/8\,H \geqq x \geqq 5/8\,H,\ \ 3/8\,H \geqq y \geqq 5/8\,H) \end{cases} \tag{97}$$

and

$$g(x,y) = 0 \quad on \quad \Gamma$$

this is shown in Fig. 28, where it is possible to see the geometry used to solve Eq. 96.

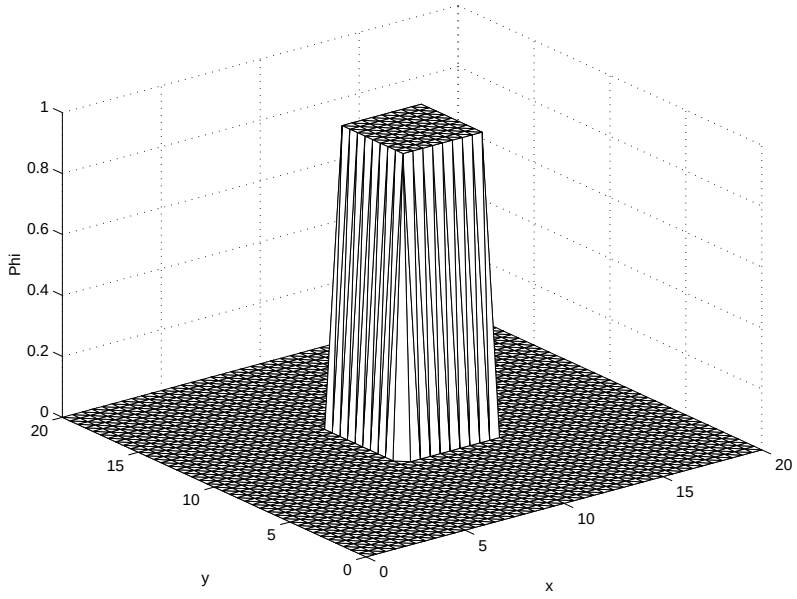

Figure 28: Function $f(x, y) = 1$ in Ω and boundary conditions $g(x, y) = 0$ on Γ .

8.1.1 Rectangular and Equi-Spaced Mesh

The first solution takes place over a rectangular mesh, where the finite difference scheme shown in Eq. 78.

The finite element is also applied in the same geometry building a triangulation. Linear elements are used for the solution[13]. Both meshes used to solve finite element and finite difference are shown in Fig. 29.

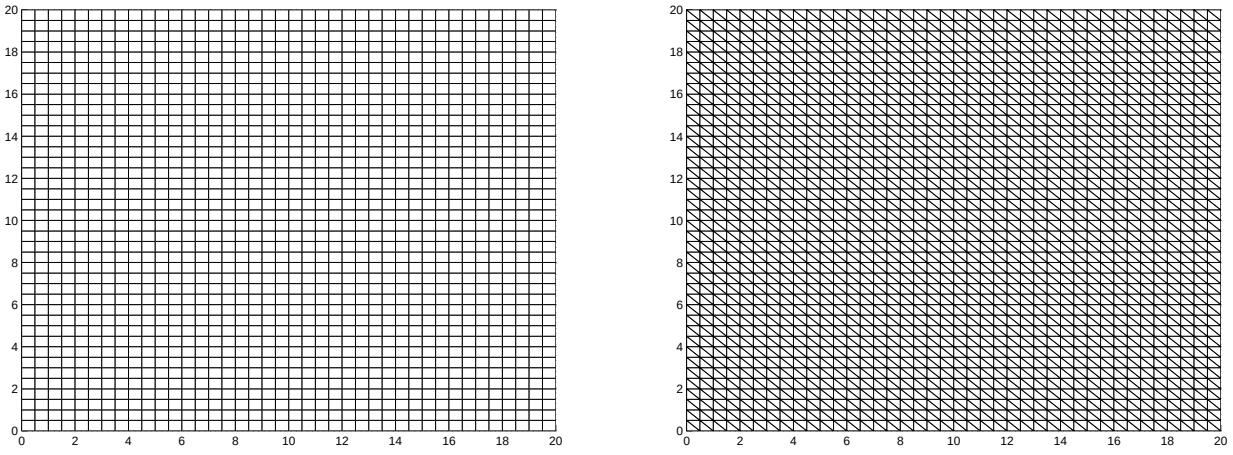

Figure 29: Meshes used to solve the finite differnce scheme and the finite element method. Number of nodes=1681, number of elements=3200.

The solution for the NNI scheme is shown in Fig. 30. As we have proved before, this solution coincides with the solution of the finite difference model.

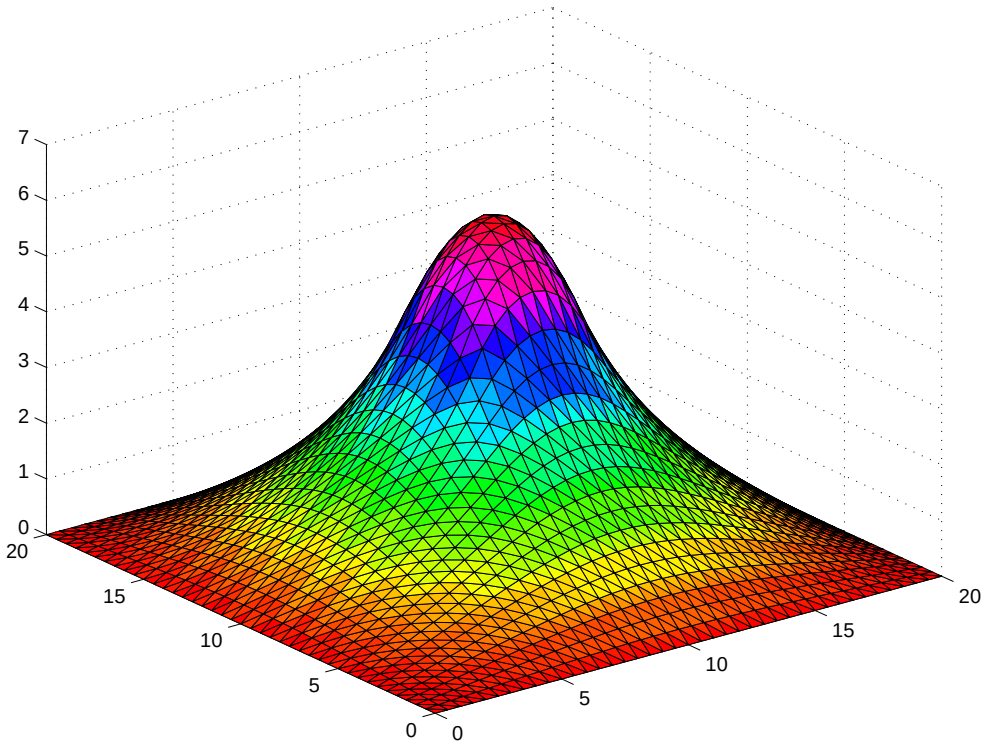

Figure 30: Solution for the NNI scheme.

The solutions are compared in the Fig. 31.

As we can see in Fig. 31 the results for the NNI and the finite difference schemes overlap and the finite element solution differs from the others. The cut that we can see was made along the transversal line shown in the figure. Comparing the approximation we deffine the norm

$$||u||_\infty = max|u|$$

where u is the solution vector. Doing this for the NNI and the finite element method we obtain the porcentual difference between the solutions:

$$\frac{||u-v||_\infty}{||u||_\infty} \times 100 = 2.68\% \tag{98}$$

where u represent the solution for the finite element method and v is the solution for the NNI approximation.

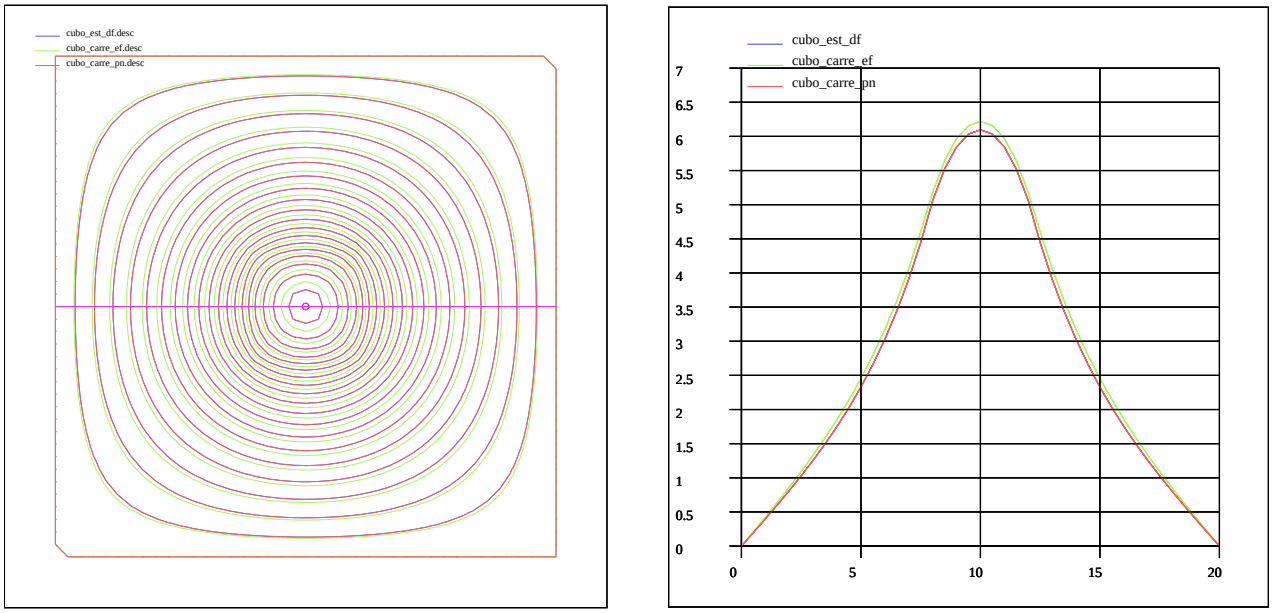

Figure 31: Solutions for the square equi-spaced mesh. Comparisons.

8.1.2 Square Domain Using a Triangulation

In this case the finite difference scheme can not be applied and therefore NNI and FEM only are compared. The discretization is shown in Fig. 32. In this figure we can also see the Voronoi diagram, not only the triangulation.

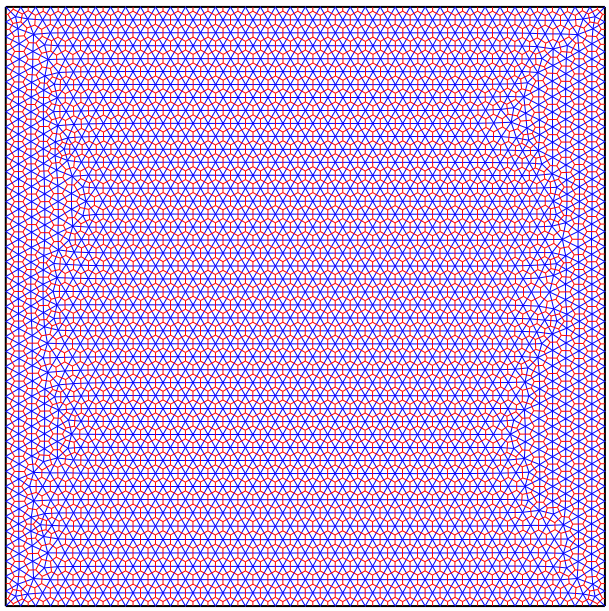

Figure 32: Second mesh used to solve the Poisson problem in the square domain. Number of nodes=1926, number of elements 3690.

The solutions are compared in Fig. 33.

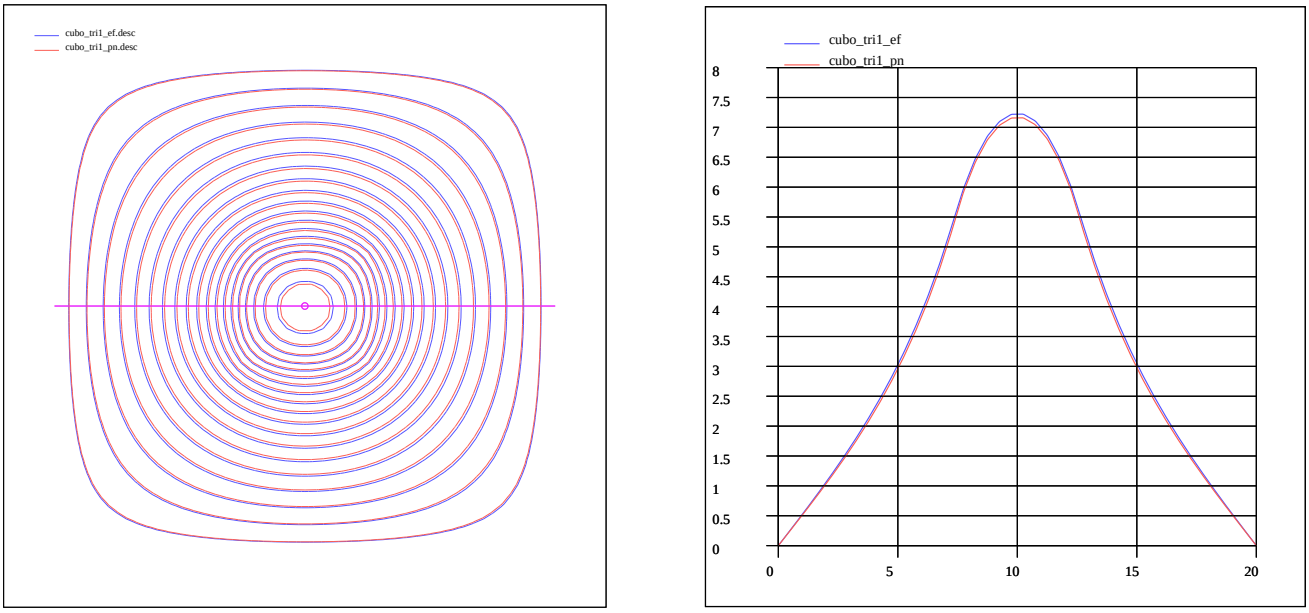

Figure 33: Solutions for the square using a triangulated mesh. Compararison.

Again we can see that the NNI solution copies the finite element solution and the value of the porcentual difference in this case is

$$\frac{||u-v||_\infty}{||u||_\infty} \times 100 = 1.27\% \tag{99}$$

The solutions for this geometry are even closer than in the case of the equi-spaced squared. Obviously the solution has been improved as the number of points used for the interpolation now is greater. This means that the support at each point is smaller. In figure Fig. 34 we can see the area of the intersection of the circumcircles, that is actually the support for the NNI method. In this figure the color indicates the area for the support in each point. In the case of an equi-spaced problem the area is the same for every point.

8.1.3 The Same Square Domain but with a Refined Triangulation in the Center

This case can be seen in Fig. 35. In this case a refinement has been done in the zone of gradient changes.

The results for this case are compared in Fig. 36.

Again we get that the two methods look alike and that the NNI method copies the finite element method. The value of the difference between the two methods for this case and appling

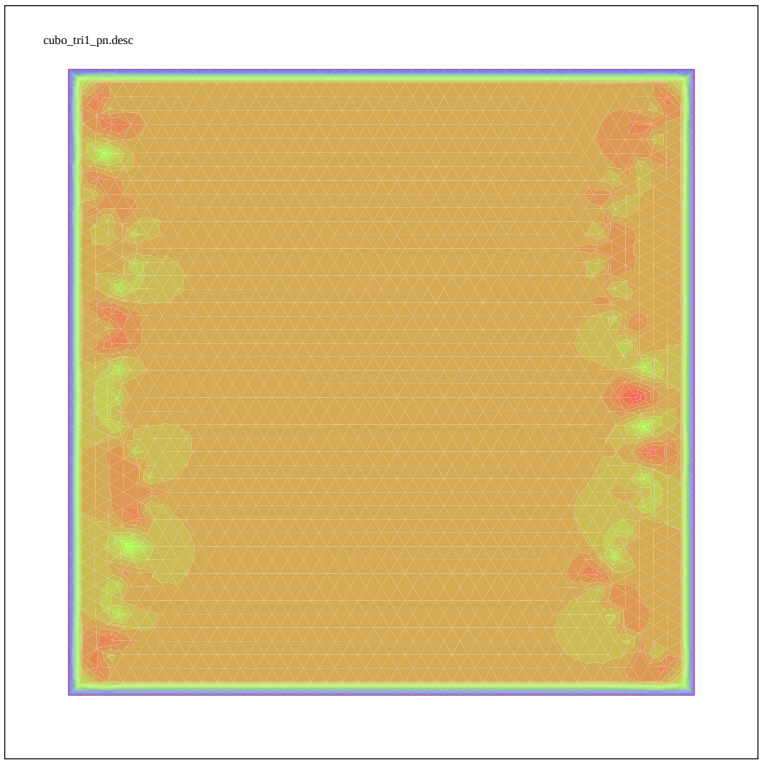

Figure 34: Support for the NNI method

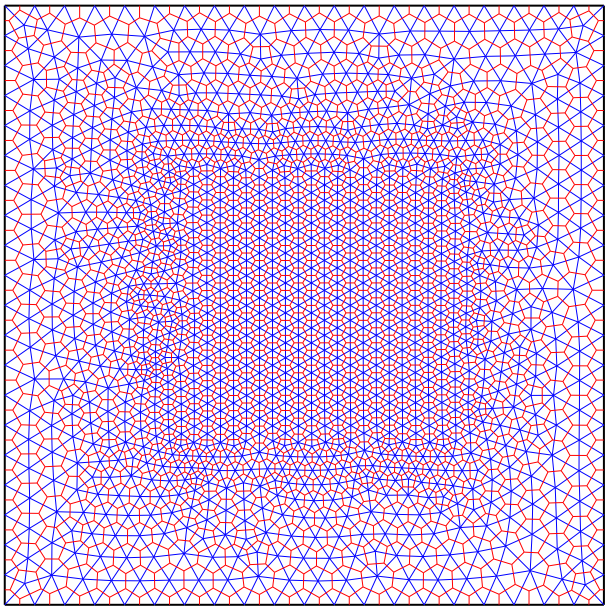

Figure 35: Squared refined mesh. Number of nodes 1142, number of elements 2202.

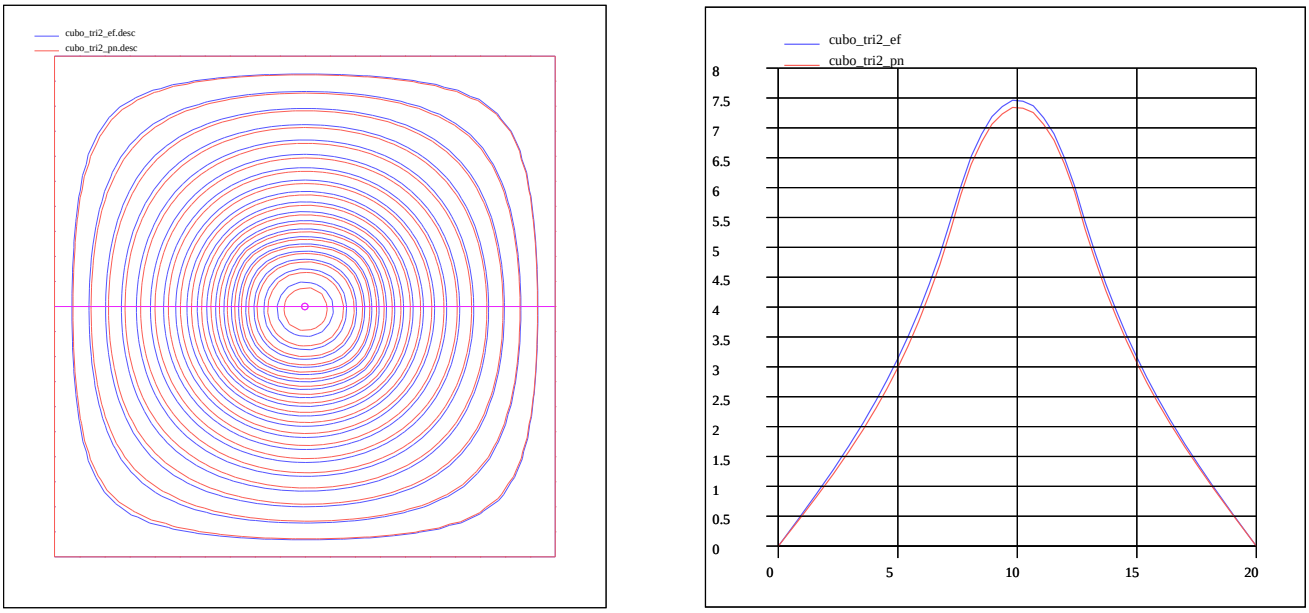

Figure 36: Solutions for the square using a triangulated refined mesh. Compararison.

the norm defined above is

$$\frac{||u-v||_\infty}{||u||_\infty} \times 100 = 2.36\% \tag{100}$$

As we can see in this case, the solutions differ a bit more now, but this result was expected as, even when we have a refinement on the center the number of nodes has decreased compared to the other case but it is still better than the first rectangular grid. This means that the methods are converging to a value and that as much as we refine the mesh the solutions will resemble even more, like in the last example, converging to the exact solution. In this geometry we have a wider variated range of areas for the support for the NNI approximation. If we look at the Fig. 37 we can see that the area in the center of the mesh, where it was refined, has a lower value than elswhere, therefore more accurate values, for the interpolation are expected at the center of the square, which is also something obvious.

8.2 Approximation of an Elliptic Problem by the Finite Element Method and the NNI method in a Nozzle Geometry

In the present section we solve an elliptic problem in a nozzle geometry.

In a two-dimensional steady, incompressible and irrotational flow, a streamfunction can be defined as a measure of the volume flow rate of fluid between a pair of streamlines. Streamlines

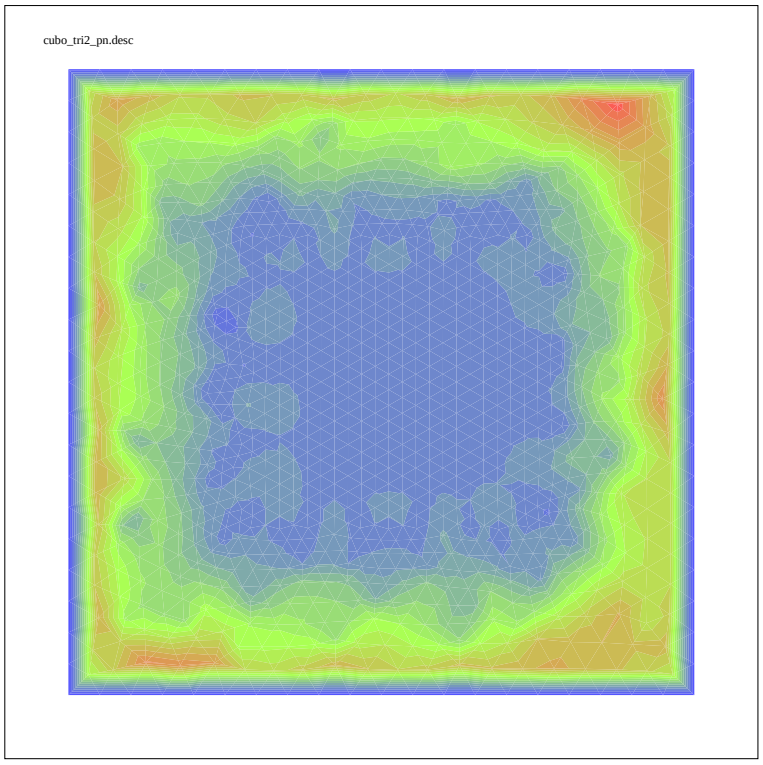

Figure 37: Support for the NNI method

are defined by joining a continuous line of points in the flow field by following the local velocity vector.

Streamlines have a constant value of streamfunction since, by their definition, all the flow must be parallel to the streamlines. No flow crosses streamlines. For two-dimensional, inviscid, incompressible flow and continuity makes the local product of distance between streamlines and velocity a constant. Thus the velocities in a flow field can be found by differenciating streamfunction with respect to the flow field coordinates y and x.

$$u = \frac{\partial \Psi}{\partial y} \quad and \quad v = -\frac{\partial \Psi}{\partial x}$$

where u is the horizontal (x-direction) component of velocity and v is the vertical (y-direction) component. Inviscid, incompressible fluid flow is called linear potential flow. The governing equations for linear potential flow are conservation of mass and the requirement that the flow be irrotational. Vorticity must be zero generally in the flow field.

$$\begin{aligned} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0 \\ \gamma = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} &= 0 \end{aligned}$$

The conservation of mass (continuity) equation can be re-written in terms of velocity potential

$$u = \frac{\partial \Phi}{\partial x} \quad and \quad v = \frac{\partial \Phi}{\partial y}$$

to produce an easily solved second order partial differential equation, the Laplace equation. Alternatively, the condition of irrotational flow can be used with the u, v velocity components being replaced by the above expressions in streamfunction gradient. This produces a Laplace equation in terms of streamfunction. This orthogonal set of equations governs the behaviour of inviscid, incompressible flow, otherwise known as linear potential flow.

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0 \quad and\,/\,or \quad \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} = 0 \tag{101}$$

This is the equation that will be solved in a geometry showed in Fig. 27. We will use two different meshes, where more points are going to be added to the second mesh.

8.2.1 First Mesh used for the Approximation

We can see this mesh in Fig. 38. In this domain the right equation of Eqs. 101 is solved. We

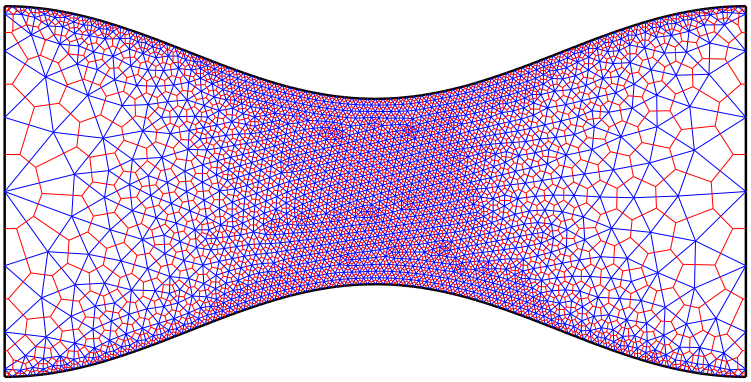

Figure 38: Geometry of the mesh used to compute the interpolation. Number of nodes=1496, number of elements 2782.

compare the results in a similar way that it was done before. A new norm used to compare the solution vectors will be defined as:

$$||u||_2 = \left[\sum_{i=1}^{n} u^2\right]^{\frac{1}{2}}$$

where n is the vector dimension and $u = \Psi_{\cdot\cdot}$. The results for this case are shown in Fig. 39. Both results are superimposed to show their similarity. Evaluating the norm described above we get

$$||u - v||_2 = 0.0311$$

where v is the vector solution for the NNI method and u is the vector solution for the FEM and $||u||_\infty = ||v||_\infty = 1_{\cdot\cdot}$.

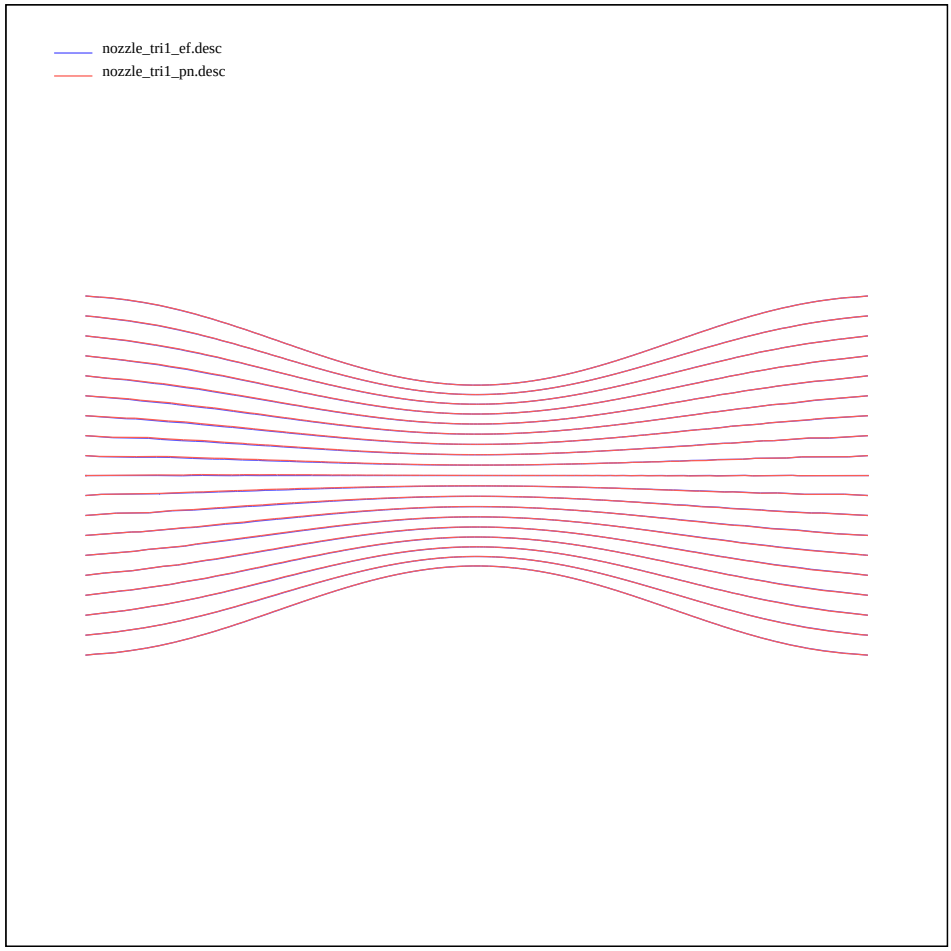

Figure 39: Stream lines for the nozzle problem.

For a more detailed comparison a cross section cut has made along the domain shown in Fig. 40.

The support area of the circumcircles used for the interpolation on each point is shown in Fig. 41. This area depends on the refinement of the mesh therefor a more accurate results are expected where the support area is smaller.

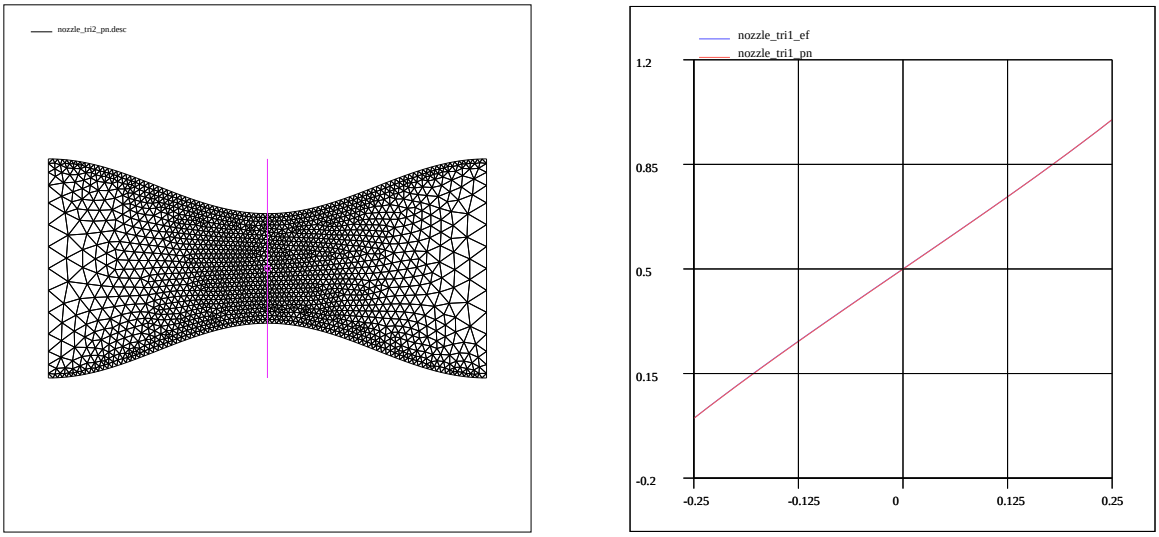

Figure 40: Coparisons in a middle perpendicular sections of the domain.

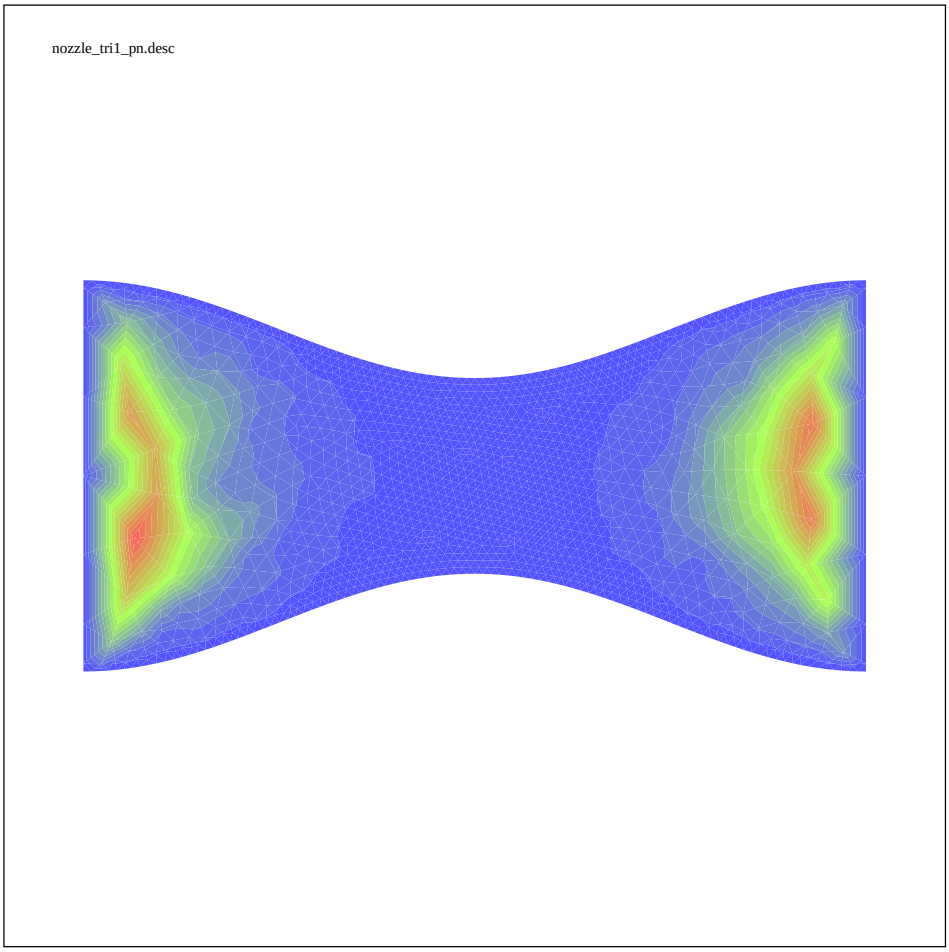

Figure 41: Support used in the NNI modeling of Laplacian.

Another figure indicates difference of the two solutions

$$e = |u - v|$$

where e is the difference between the solutions. This value is plotted in Fig. 42.

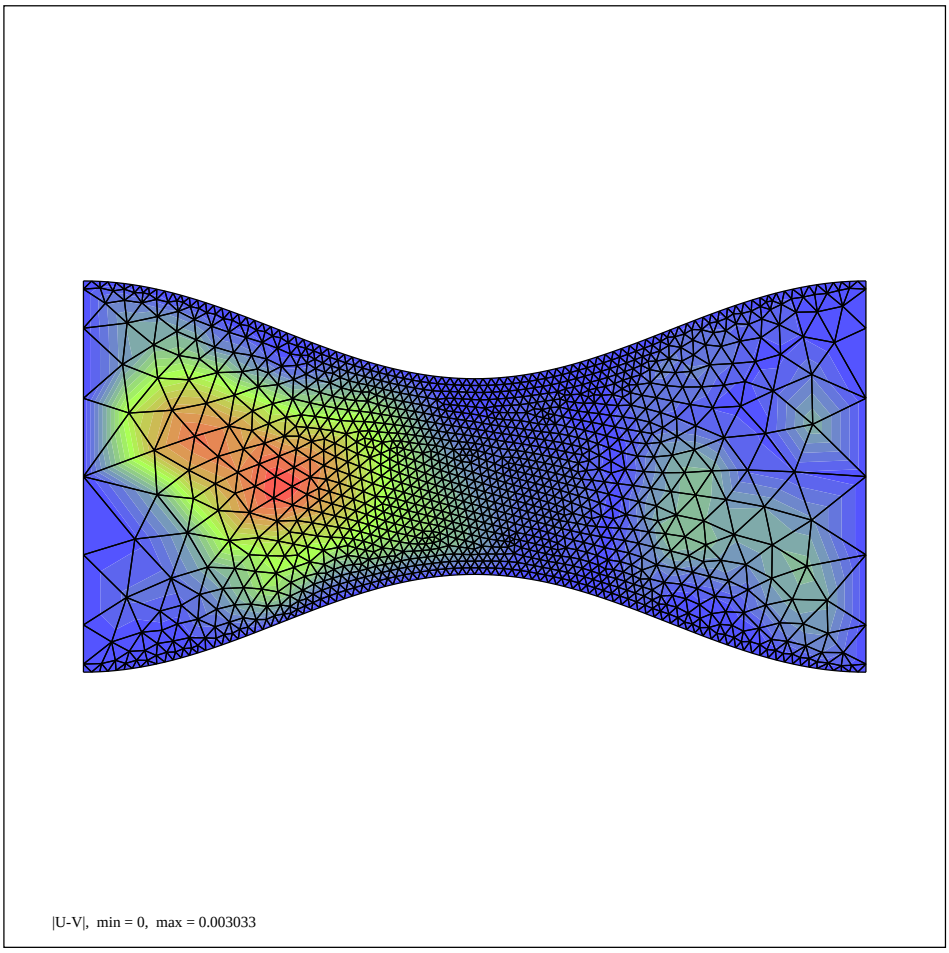

Figure 42: Difference between the approximation by FEM and the approximation by NNI.

8.2.2 Second Mesh used for the Approximation

The mesh used in this case is plotted in Fig. 43. As we can see, a refinement was made on the areas closer to the ends of the nozzle. In this domain, Eq. 101 was solved. We compare the results using the same norm defined above.

$$||u - v||_2 = 0.0294$$

where v is the vector solution for the NNI method and u is the vector solution for the FEM. As we can see, and as we expected to see, both solutions get closer when the mesh is refined. It is

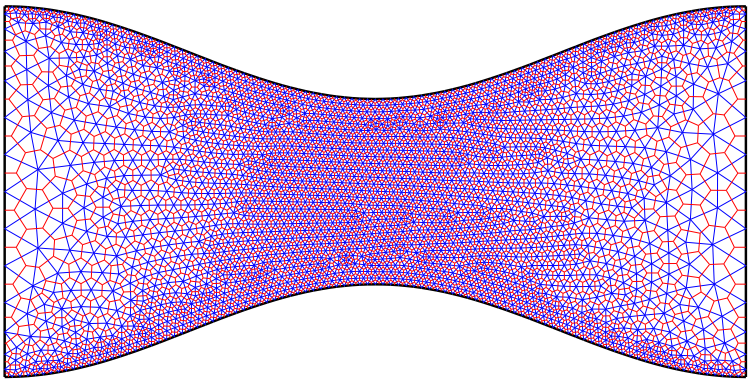

Figure 43: Geometry of the mesh used to compute the interpolation. Both the Delaunay triangulation and the Voronoi diagram are shown. Number of nodes=1986, number of elements=3750.

expected that as the mesh is refined, the closer the solutions are and eventually the solutions converge to the exact solution.

The results for this case are shown in Fig. 44. Both results are superimposed to show their similarity.

In Fig. 45 a transversal section cut is shown and again the solutions are superimposed.

The support area of the circumcircles used for the interpolation with a more refined mesh is shown in Fig. 46.

A last figure shows the difference of the two solutions computed as:

$$e = |u - v|$$

where e is the difference between the solutions. This value is plotted in Fig. 47.

For both cases, the standard and the refined, it is possible to see a strong dependence of this difference with respect to the mesh geometry. While the shape of the geometry and the equations solved are both symmetric, the difference between the solutions are not. This means that a different shape of the triangulation and distribution of vertices gives, as a result, different local solutions. On the other hand, it is not possible to say which of the two solutions is more accurate.

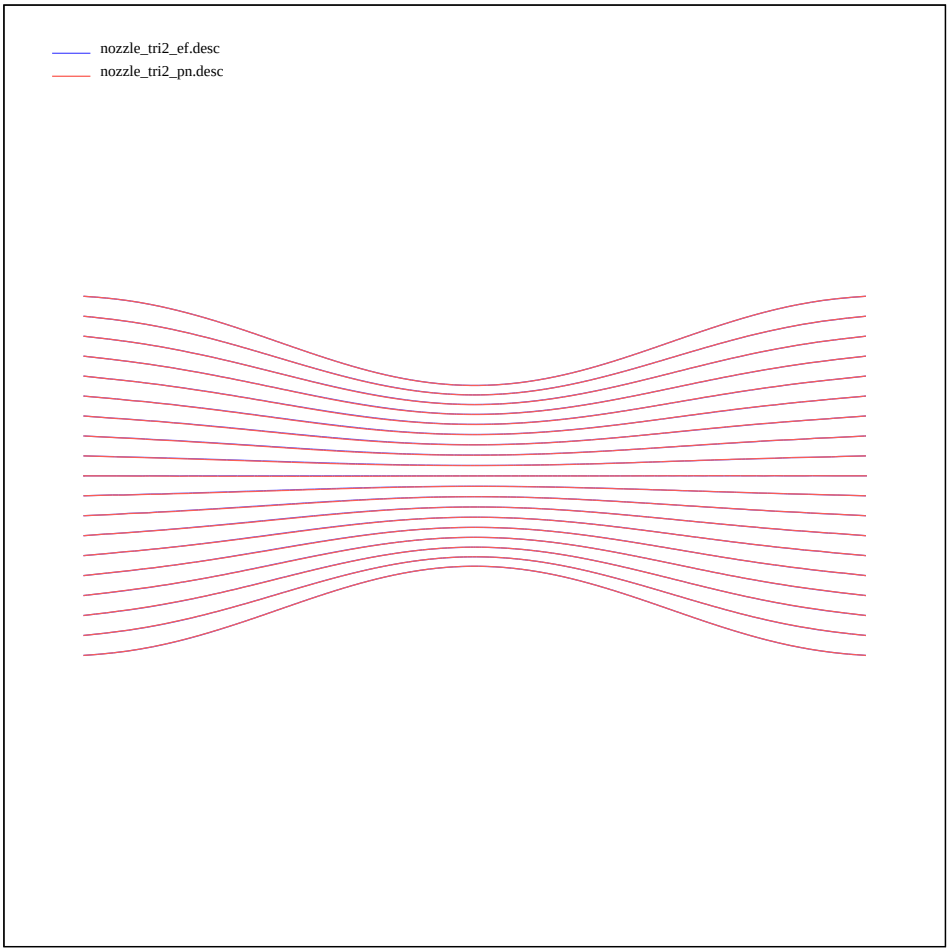

Figure 44: Stream lines for the nozzle problem.

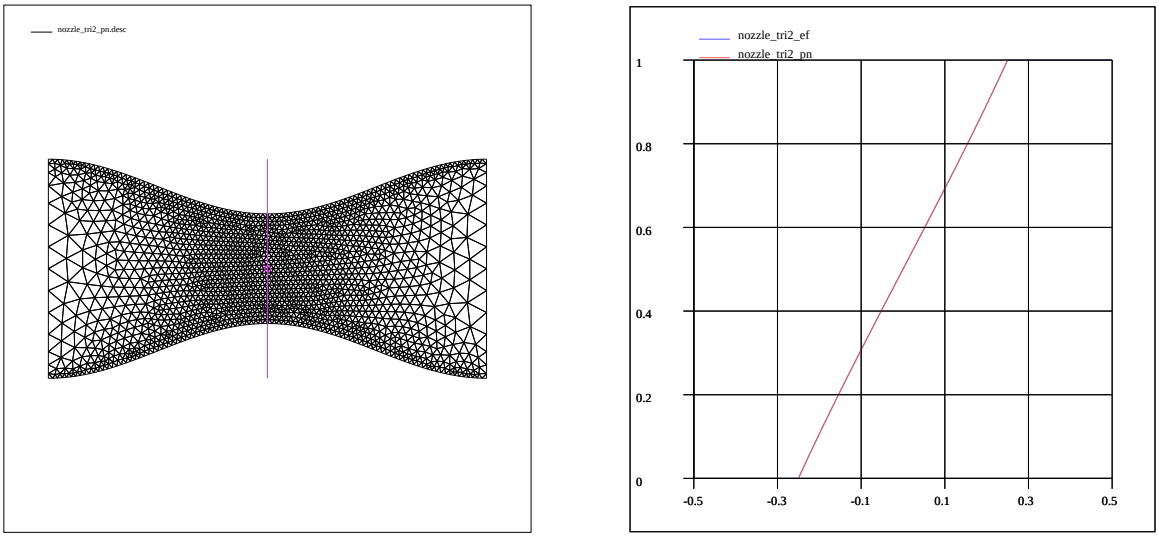

Figure 45: Comparisons in a middle perpendicular sections of the domain.

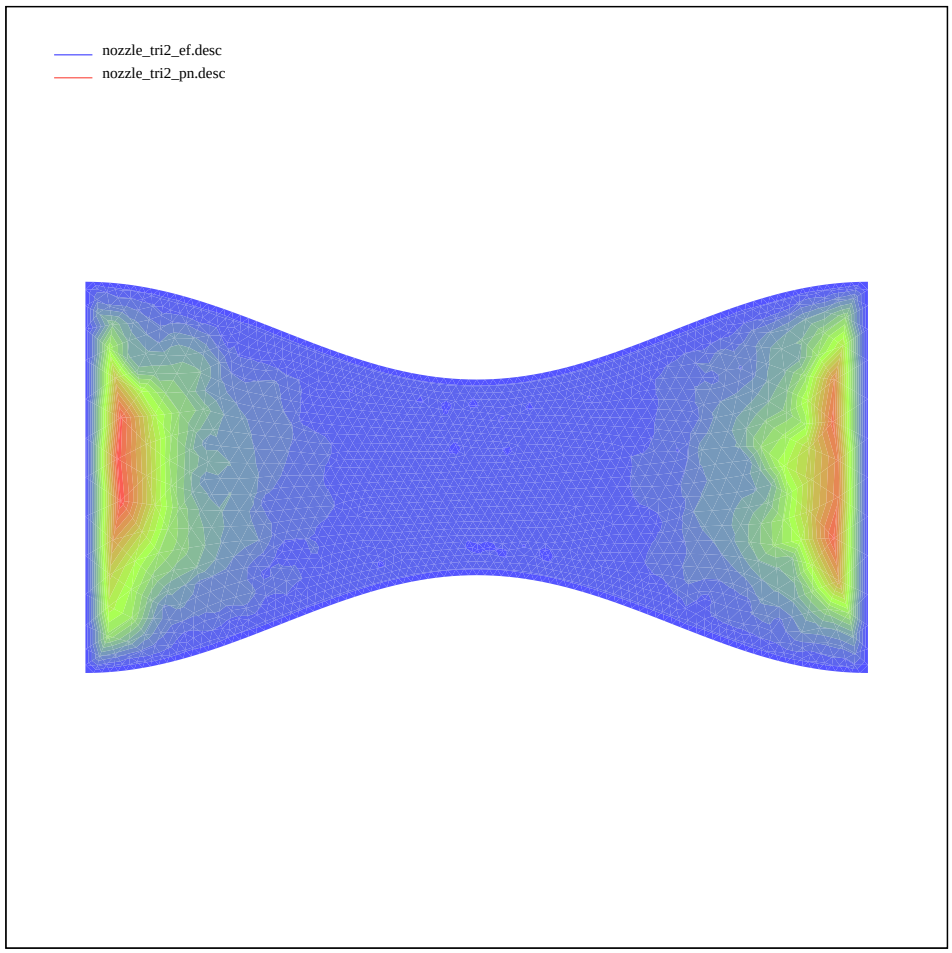

Figure 46: Support used in the NNI modeling of Laplacian.

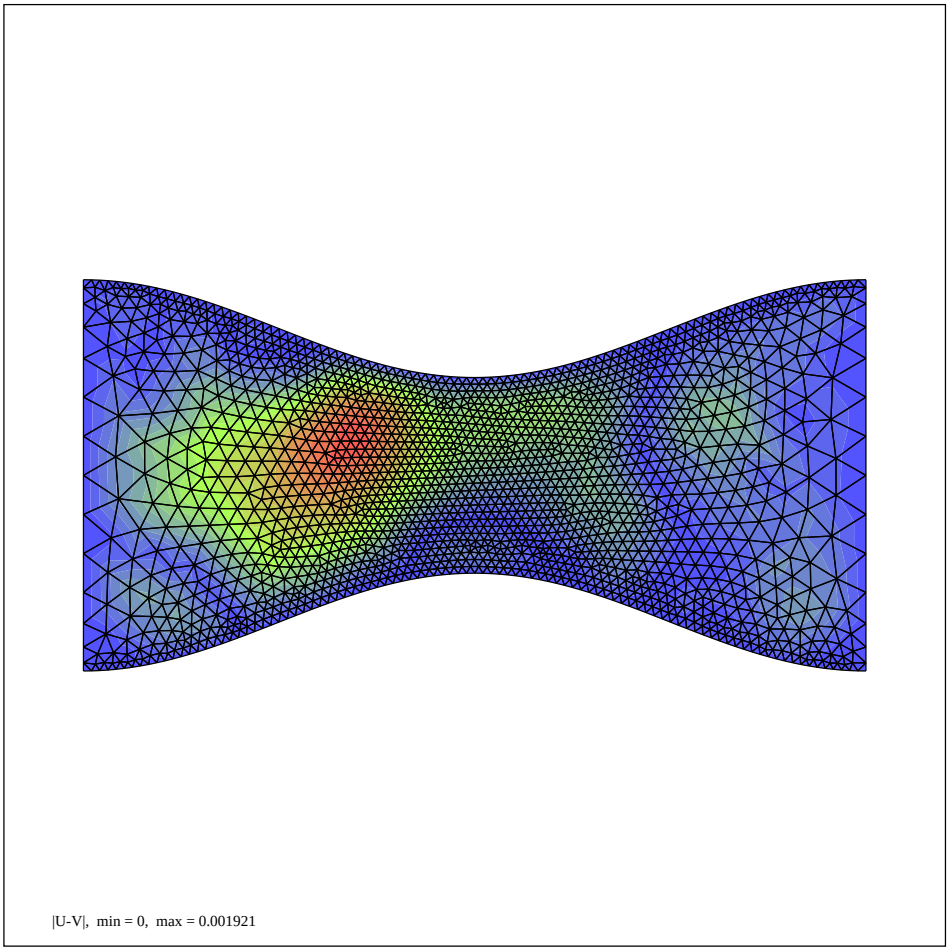

Figure 47: Difference between the approximation by FEM and the approximation by NNI.

9 Conclusions

After analyzing and comparing numerous numerical results in which Laplace and the Poisson equations were solved in different domains, we can say the the approximation made by natural neighbors for solving partial differential equations converges to the exact solution. The order of convergence is known with precision for the rectangular and equi-spaced grid. For that case a second order of convergence was proved. In the other cases the results are empirical, but the method follows the trend of classical FEM guaranteeing a second order accuracy. So it is expected that the solution applying the natural neighbor interpolation has the same order of convergence. The method adaptes very easily to the change in geometry and also to mesh refinement. As was mentioned in the introduction, there is no geometric restriction with respect to the shape of the domain. Further examples should be done in more complicated geometries to establish this. Eventually a 3D geometry would not represent a major complication.

The NNI method has proved to have a very good efficiency compared to the more traditional methods with an easy implementation. It still has to be proved in more complicated problems. The perspectives that it presents, as an alternative to the traditional methods, is good and as a mesh-less method, the range of problems that it could solve is wide. Future applications will give a more precise idea of the potential of the NNI approximation.

10 APPENDIX A : The code to solve the interpolation

```
// ========================================================================
//
// Code to compute the natural neighbor interpolation and its application
// solving partial differential equation.
//
// ------------------------------------------------------------------------
// Cases studied Laplace and Poisson. Any geometry and dirichlet boundary
// conditions.
// ========================================================================



#include <CGAL/basic.h>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <iomanip.h>
#include <fstream>
#include <strstream>
#include <vector>
#include <CGAL/Cartesian.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Triangulation_euclidean_traits_2.h>
#include "Natural_neighbors_2.h"
#include <CGAL/Circle_2.h>
#include <CGAL/IO/Window_stream.h>

extern "C" void dgmres_
   (int *nt, double g[],double phi[],int *nelt,int ia[],int ja[],double  sk_sol[], int
*isym,int *itol, double *tol, int *itmax, int *iter, double *err, int *ierr, int *iunit,
```

```
double sb[],double sx[],double rgwk[],int *lrgw,int igwk[],int *ligw, double rwork[],int
iwork[]);

typedef double Coord_type;

typedef CGAL::Cartesian<Coord_type>  Rp;

typedef CGAL::Point_2<Rp>  Point;
typedef CGAL::Circle_2<Rp>  Circle;
typedef CGAL::Vector_2<Rp> Vector;
typedef CGAL::Segment_2<Rp>  Segment;
typedef CGAL::Ray_2<Rp>  Ray;
typedef CGAL::Line_2<Rp>  Line;
typedef CGAL::Triangle_2<Rp>  Triangle;

#include "New_vertex_base.h"

typedef CGAL::Triangulation_euclidean_traits_2<Rp> Gt;
typedef New_vertex_base<Gt> Vb;
typedef CGAL::Triangulation_face_base_2<Gt>  Fb;
typedef CGAL::Triangulation_default_data_structure_2<Gt,Vb,Fb> Tds;
typedef CGAL::Natural_neighbors_2<Gt,Tds> Nn;

typedef   Nn::Face  Face;
typedef   Nn::Vertex Vertex;
typedef   Vertex::Vertex_handle Vertex_handle;
typedef   Face::Face_handle Face_handle;



typedef   Nn::Face_circulator  Face_circulator;
```

```
typedef   Nn::Vertex_circulator  Vertex_circulator;

typedef   Nn::Locate_type Locate_type;

typedef   Nn::Face_iterator  Face_iterator;
typedef   Nn::Vertex_iterator  Vertex_iterator;
typedef   Nn::Edge_iterator  Edge_iterator;
typedef   Nn::Line_face_circulator  Line_face_circulator;
typedef   Nn::Edge_circulator  Edge_circulator;
typedef   Nn::Natural_output out;


#include "Natural_Neighbors_Interpolation.h"

#include "Local_triangulation.h" //triangulacion local para la interpolacion




int main()
{


  Nn dt;
  Nn::Vertex_iterator it;
  Nn::Face_iterator fc;
  /* ------------------------------------------------------ */
  /*--------------- Data entry ---------------------------- */
  /* ------------------------------------------------------ */

 //Lee nuevos Puntos desde un archivo y pone los vertices en un vector
```

```
  ifstream datain("./input/malla.dat");
  double dx;
  int nt;
  datain >>nt;
  Vertex_handle* const vertices = new Vertex_handle[nt];
  Point p;
  Nn::Vertex_handle vh;
  int numeracion;
  int num_bc=0;


  for( int i=0; i<nt ; i++)
    {
      int bound_cond;
      datain >>numeracion >>p>> bound_cond ;
      vh=dt.insert(p);
      vh->label=numeracion+1;
      vh->boundary_c=bound_cond;
      if(bound_cond!=0)num_bc++;
      vertices[i]=vh;
    }

  int npbound[num_bc];
  int cont_bound=0;
  for( int i=0; i<nt ; i++)
    {
      if(vertices[i]->boundary_c!=0)
{
  npbound[cont_bound]=vertices[i]->label;
  cont_bound++;
```

```
}
    }

  //-------------------------------------------------
  //Method used for the solution
  //-------------------------------------------------
  int metodo,problema,prob_lap_pois,fuente,estruc;
  ifstream in("./input/dat.in");
  in >>metodo>>dx>>problema>>prob_lap_pois>>fuente>>estruc;

  if(estruc==0)cout <<"malla estructurada"<<endl;

  if(estruc!=0)cout <<"malla no estructurada"<<endl;

  if(metodo==0 && problema==1)
    {
      cout <<"solucion meditante Dif finitas del cuadrado"<<endl;
    }
  if(metodo==0 && problema==2)
    {
      cout <<"solucion meditante Dif finitas del nozzle"<<endl;
    }
  if(metodo==1 && problema==1)
    {
      cout <<"solucion meditante ptos naturales del cuadrado"<<endl;
    }
  if(metodo==1 && problema==2)
    {
      cout <<"solucion meditante ptos naturales del nozzle"<<endl;
    }
```

```
  if(problema==1)
    {
  //-------------------------------------------------
  //-----Boundary conditions for the squared
  //-------------------------------------------------

  double phi_1,phi_2,phi_3,phi_4;
  cout <<"borde inferior"<<endl;
  cin >>phi_1;
  cout <<" "<<endl;

  cout <<"borde derecho"<<endl;
  cin >>phi_2;
  cout <<" "<<endl;

 cout <<"borde superior"<<endl;
  cin >>phi_3;
  cout <<" "<<endl;

 cout <<"borde izquierdo"<<endl;
  cin >>phi_4;
  cout <<" "<<endl;




  for( int i=0; i<nt ; i++)
    {
      vertices[i]->phi=0.0;
      if(vertices[i]->boundary_c!=0)
{
  if(vertices[i]->boundary_c==1)
```

```
    {
      vertices[i]->phi=phi_1;
    }
  if(vertices[i]->boundary_c==2)
    {
      vertices[i]->phi=phi_2;
    }
  if(vertices[i]->boundary_c==3)
    {
      vertices[i]->phi=phi_3;
    }
  if(vertices[i]->boundary_c==4)
    {
      vertices[i]->phi=phi_4;
    }
}
    }
    }

  if(problema==2)
    {
      //-------------------------------------------------
      //-----Boundary conditions for the Nozzle
      //-------------------------------------------------
      cout <<"cond lineal en los extremos"<<endl;
      cout <<"ingresar el valor de Phi en la parte superior: "<<endl;
      double phi_n;
      cin>>phi_n;


 for( int i=0; i<nt ; i++)
```

```
  {
    vertices[i]->phi=phi_n*vertices[i]->point().y()+phi_n/2.0;
    if(vertices[i]->boundary_c!=0)
      {
if(vertices[i]->boundary_c==1)
  {
    vertices[i]->phi=phi_n*vertices[i]->point().y()+phi_n/2.0;;
  }
if(vertices[i]->boundary_c==2)
  {
    vertices[i]->phi=phi_n*0.5+phi_n/2.0;;
  }
if(vertices[i]->boundary_c==3)
  {
    vertices[i]->phi=0.0;
  }
      }
  }

   }
 //-----------------------------------------------------------------
 // Initial conditions to solve finite elements in matlab
 //-----------------------------------------------------------------


 ofstream bound_cond("./output/bound_cond.bc");

 bound_cond <<num_bc<<endl;


 for(int i=0;i<cont_bound;i++)
```

```
   {
     bound_cond <<npbound[i]<<endl;
   }


for(int i=0;i<cont_bound;i++)
  {
    bound_cond <<vertices[npbound[i]-1]->phi<<endl;
  }


 /* ------------------------------------------------------ */
 /* -------------- Inicialization            ------------ */
 /* ------------------------------------------------------ */


 int const num_vec = 80;
 int const num_part = nt;

 double sk[num_part][num_vec];
 double g[num_part];
 int num[num_part][num_vec];
 int npp[num_part];

 //Solver

 int isym=0;
 int itol=1;
 double tol=0.0000001;
 double rwork[10];
 int iwork[10];
```

```
int iter;
double err;
int ierr;
int iunit=0;
int ligw=30;
int maxl=10;
int itmax=maxl*(10+1);
int lrgw=1+nt*(maxl+6)+maxl*(maxl+3)+50;
double rgwk[lrgw];
int igwk[ligw];
    igwk[1]=maxl;
    igwk[2]=maxl;
    igwk[3]=0;
    igwk[4]=0;
    igwk[5]=10;
    igwk[6]=lrgw-50;
    igwk[7]=5;

double sb[num_part];
double sx[num_part];
double sk_sol[num_part*num_vec];
int ia[num_part*num_vec];
int ja[num_part*num_vec];
double phi[num_part];



for(int i = 0 ; i<num_part ; i++)
  {
    phi[i]=vertices[i]->phi;
  }
```

```
  /*-------------------------------------------------*/
  //--- Definition of the matrix to solve the laplacian-
  /*-------------------------------------------------*/

  for(it=dt.vertices_begin();it!=dt.vertices_end();it++)
    {
      if(it->boundary_c==0)
{
  interpolation(it,dt);
}
      else
{
  it->num_vecinos=0;
  it->suma_func=-1.0;
}
    }


  //-------------------------------------------------
  /*---------Itaration over all the points------------*/
  //-------------------------------------------------

  ofstream gout("./output/g");

  for(int i=0 ;  i < nt; i++)
    {
      double g_const;
      // ------- sparse matrix  and natural neighbors

      sk[i][0]= -vertices[i]->suma_func ;
```

```
      num[i][0]=vertices[i]->label;

      //------ Number of neighbors on each point

      npp[i]=vertices[i]->num_vecinos;

      for(int j=1 ; j < npp[i]+1 ;j++)
{
  sk[i][j]=vertices[i]->h_kernel[j] ;
  num[i][j]=vertices[i]->num[j];
}
      // -------

      /* ---Independent term--- */
      if(prob_lap_pois==1)
{
  if(metodo==1)
    {
      g_const=vertices[i]->lambda*vertices[i]->suma_func/2.0/2.0;
    }
  else
    {
      g_const=dx*dx/4.0;
    }

  //Diferent kind of surce term for the poisson eq
  if(fuente==1)
     {
       // 2.71....**(1/((10*x-5)**2+(10*y-5)**2))

       double divid=(pow((5.0*vertices[i]->point().x()-50.0),2.0)+
```

```
     pow((5.0*vertices[i]->point().y()-50.0),2.0));
       if(divid==0)divid=0.000001;
       g[i]=-(pow(2.7182818,(1.0/divid)))*g_const;
       gout <<(pow(2.7182818,(1.0/divid)))<<endl;

     }
  if(fuente==2)
    {
      if(vertices[i]->point().x()>7.5 && vertices[i]->point().y()>7.5 &&
vertices[i]->point().x()<12.5 && vertices[i]->point().y()<12.5)
{
  g[i]=-1.0*g_const;
  gout <<1.0<<endl;
}
      else
{
  g[i]=0.0;
  gout <<0.0<<endl;
}
    }
}
  if(prob_lap_pois==2)
    {
      g[i]=0.0;
      gout <<0.0<<endl;
    }

 // -------
       }

  /*----------------- end of the iteration over all the points---*/
```

```
  //-----------------------------------------------------------
  // Boundary conditions over the matrix
  //-----------------------------------------------------------


  for(it=dt.vertices_begin();it!=dt.vertices_end();it++)
    {
      if(it->boundary_c!=0)
{
  for (int i=1; i<it->num_vecinos+1; i++)
    {
      sk[it->label-1][i]=0.0;
    }
  sk[it->label-1][0]=-1.0;
  g[it->label-1]=-phi[it->label-1];
}
    }


  //-----------------------------------------------------------
  //        Preconditioning for the solver
  //-----------------------------------------------------------

  int nelt=0;

  for(int i=0;i<nt;i++)
    {
    for(int j=0 ; j < npp[i]+1 ;j++)
   {
```

```
     sk_sol[nelt]=sk[i][j];
     ia[nelt]=i+1;
     ja[nelt]=num[i][j];
     nelt++;
   }
    }

  //-------------------------------------------------------------
  // Printing the matrix to solv it with matlab for correlation
  //-------------------------------------------------------------

  ofstream matriz("./output/matriz");


  for(int i=0;i<nelt;i++)
    {
      matriz <<ia[i]<<" "<<ja[i]<<" "<<sk_sol[i]<<endl;
    }

  //-------------------------------------------------------------
  // Solver for sparse matrix
  //-------------------------------------------------------------

  dgmres_
    (&nt,  g,phi,&nelt, ia, ja,sk_sol,  &isym,   &itol,  &tol, &itmax, &iter, &err,
&ierr, &iunit, sb, sx, rgwk,&lrgw, igwk,&ligw,  rwork,  iwork);




  for(int i=0; i<nt; i++)
```

```
    {
      vertices[i] -> phi = phi[i];
    }



  //------------------------------------------------------------
  //  Gradients computing
  //------------------------------------------------------------


  Vector cero(0.0,0.0);
  Nn::Vertex_circulator circ,done;

  for(int i=0;i<nt;i++)
    {
     circ=dt.incident_vertices(vertices[i]);
     done=circ;
     double min_phi=circ->phi;
     do{
       if(circ->phi < min_phi) min_phi = circ -> phi;
       circ++;
     }while(circ!=done);
      vertices[i]->grad=cero;
      //Only compute the gradient if the point does not belong to the convex hull
      if(dt.is_infinite(vertices[i])==0)
{
  for(int j=1;j<npp[i]+1;j++)
    {
      vertices[i]->grad=vertices[i]->grad+2.0/vertices[i]->suma_func*
(vertices[num[i][j]-1]->phi-min_phi)/
```

```
(squared_distance(vertices[num[i][j]-1]->point(),
  vertices[i]->point()))*
(vertices[num[i][j]-1]->point()-vertices[i]->point())*
vertices[i]->h_kernel[j];

    }
  if(problema==2)
    {
      // Gradient to compute the velocity for the laplace equation
      Vector menos(vertices[i]->grad.x()*(-2.0),0.0);
      vertices[i]->grad=vertices[i]->grad+menos;
      Vector vel(vertices[i]->grad.y(),vertices[i]->grad.x());
      vertices[i]->grad=vel;
    }
}
    }

  //-------------------------------------------------------
  //-------------------------------------------------------




  ofstream vigie("./output/vigie.dat");
  ofstream x("./output/x");
  ofstream y("./output/y");
  ofstream z("./output/z");
  ofstream k("./output/k");
  ofstream grad("./output/grad");

  vigie <<"points "<<nt<<endl;
```

```
  for(int i=0;i<nt;i++)
    {
      x <<vertices[i]->point().x()<<endl;
      y <<vertices[i]->point().y()<<endl;
      z <<vertices[i]->phi<<endl;
      k <<vertices[i]->lambda*(M_PI+1.37/2.0)<<endl;
      //k <<g[i]<<endl;
      grad <<vertices[i]->point()<<" "<<vertices[i]->grad<<endl;
      vigie <<vertices[i]->point()<<endl;
    }


  ofstream tri("./output/tri");
  ofstream conect("./output/conect.e");

  int count_elem=0;
  // Orinting of the triangulation for matlab
  for(fc=dt.faces_begin();fc!=dt.faces_end();fc++)
    {
      for (int i=0;i<3;i++)
{
  tri <<fc->vertex(i)->label<<" ";
}
      tri <<"\n";
      count_elem++;
    }

  // conectividades para matlab
    conect <<count_elem<<endl;
    for(fc=dt.faces_begin();fc!=dt.faces_end();fc++)
      {
```

```
conect <<0<<" ";
for (int i=0;i<3;i++)
  {
    conect <<" "<<fc->vertex(i)->label-1<<" ";
  }
conect <<0<<" "<<0<<" "<<0<<" "<<0<<" "<<0<<" "<<0<<" "<<0.001<<" "<<0.0001<<"
"<<0<<endl;
      }

  int num_el;
  if(problema==1 && estruc==0)
    {

      vigie <<"triangles "<<count_elem<<endl;
      for(fc=dt.faces_begin();fc!=dt.faces_end();fc++)
{
  for (int i=0;i<3;i++)
    {
      vigie <<" "<<fc->vertex(i)->label-1<<" ";
    }
  vigie <<" "<<endl;
}

    }
 if(problema==1 && estruc==1)
   {
     ifstream grid("./input/vigie.e");
     grid >>num_el;
     vigie <<"triangles "<<num_el<<endl;

     for(int i=0;i<num_el;i++)
```

```
      {
int n1,n2,n3,n00;
double n0;
grid >>n00>>n1>>n2>>n3>>n00>>n00>>n00>>n00>>n00>>n00>>n0>>n0>>n00;
vigie <<n1<<" "<<n2<<" "<<n3<<endl;
      }

  }

if(problema==2 && estruc==1)
  {
    ifstream grid("./input/vigie.e");
    grid >>num_el;
    vigie <<"triangles "<<num_el<<endl;

    for(int i=0;i<num_el;i++)
      {
int n1,n2,n3,n00;
double n0;
grid >>n00>>n1>>n2>>n3>>n00>>n00>>n00>>n00>>n00>>n00>>n0>>n0>>n00;
vigie <<n1<<" "<<n2<<" "<<n3<<endl;
      }

  }



 vigie <<"scalars Phi"<<endl;
 for(int i=0;i<nt;i++)
   {
     vigie <<vertices[i]->phi<<endl;
```

```
    }
  if(problema==2)
    {
      vigie <<"scalars Pressure"<<endl;
      for(int i=0;i<nt;i++)
{
  // Rho=1 Bernoulli
  vigie <<0.5*(vertices[i]->grad.x()*vertices[i]->grad.x()+
       vertices[i]->grad.y()+vertices[i]->grad.y())<<endl;
}
    }


  vigie <<"scalars Area"<<endl;
  for(int i=0;i<nt;i++)
    {
      vigie <<vertices[i]->soporte<<endl;
    }
  vigie <<"vectors Vel Velx Vely .1"<<endl;
  for(int i=0;i<nt;i++)
    {
      vigie <<vertices[i]->grad<<endl;
    }



    // Prints on screen the biggest value of Phi
    double phi_max=0.0;
    for(int i=0;i<nt;i++)
      {
if(vertices[i]->phi>phi_max)phi_max=vertices[i]->phi;
```

```
      }
    cout <<phi_max<<endl;

}


// ============================================================================
//
// Copyright (c) 1997 The CGAL Consortium
//
// This software and related documentation is part of an INTERNAL release
// of the Computational Geometry Algorithms Library (CGAL). It is not
// intended for general use.
//
// ----------------------------------------------------------------------------
//
// release       : $CGAL_Revision: CGAL-2.0-I-20 $
// release_date  : $CGAL_Date: 1999/06/02 $
//
// file          : include/CGAL/Natural_neighbors_2.h
// package       :
// source        : $RCSfile: Natural_neighbors_2.h,v $
// revision      : $Revision: 1.6 $
// revision_date : $Date: 2000/05/11 13:12:21 $
// author(s)     : Tran Kai Frank DA <Frank.Da@sophia.inria.fr>
//
// coordinator   : INRIA Sophia-Antipolis (<Mariette.Yvinec@sophia.inria.fr>)
//
// ============================================================================

#ifndef NATURAL_NEIGHBORS_2_H
```

```
#define NATURAL_NEIGHBORS_2_H

#include <vector>
#include <math.h>

#include <CGAL/determinant.h>
#include <CGAL/squared_distance_2.h>
#include <CGAL/Delaunay_triangulation_2.h>

//--------------------------------------------------------------------
CGAL_BEGIN_NAMESPACE
//--------------------------------------------------------------------

template < class Gt, class Tds >
class Natural_neighbors_2 : public Delaunay_triangulation_2<Gt, Tds>
{
private:

  typedef typename Gt::Rep Rp;
  typedef typename Rp::FT Coord_type;
  typedef pair< Vertex_handle, Coord_type> Natural_data;

public:

  typedef vector<Natural_data> Natural_output;

  //------------------------- CONSTRUCTORS ----------------------------


  Natural_neighbors_2()
    : Delaunay_triangulation_2<Gt, Tds>() {}
```

```
  Natural_neighbors_2(const Gt& gt)
    : Delaunay_triangulation_2<Gt, Tds>(gt) {}

  Natural_neighbors_2(const Vertex_handle& v, const Gt& gt=Gt())
    : Delaunay_triangulation_2<Gt, Tds>(v, gt) {}

  Natural_neighbors_2(const Triangulation_2<Gt,Tds> &tr)
    : Delaunay_triangulation_2<Gt, Tds>(tr) {}


  //----------------------- OPERATIONS ---------------------------------

  bool test_conflict(const Face_handle& fh, const Point& p) const;

  bool find_conflicts(const Point& p, list<Edge>& le,
      bool& already, Face_handle hint = Face_handle()) const;

  Natural_output natural_areas(const Point& p, bool& res, double& suma_area) const;

  double suma_areas(const Point& p, bool& res) ;

};

//------------- Template Member Functions ------------------------------

template < class Gt, class Tds >
bool
Natural_neighbors_2<Gt,Tds>::test_conflict(const Face_handle& fh,
  const Point& p) const
{
```

```
  return ((side_of_oriented_circle(fh,p) != ON_NEGATIVE_SIDE ));
}

//------------------------------------------------------------------

template < class Gt, class Tds >
bool
Natural_neighbors_2<Gt,Tds>::find_conflicts(const Point& p, list<Edge>& le,
   bool& already,  Face_handle hint) const
{
  bool result(false);
  // sets in le the counterclocwise list of neighbors (edges)
  // of the destroyed region when  $p$ is  inserted
  Face_handle fh= locate(p, hint);
  Vertex_handle vh;

  if (is_infinite(fh))
    vh = nearest_vertex(p);
  else
    vh = nearest_vertex(p, fh);

  if ((CGAL::compare_x(vh->point(), p) == EQUAL)&&
      (CGAL::compare_y(vh->point(), p) == EQUAL))
    already=true;
  else
  {
    already=false;

    if (!is_infinite(fh))
      {
le.push_back(Edge(fh->neighbor(0),fh->neighbor(0)->index(fh)));
```

```
le.push_back(Edge(fh->neighbor(1),fh->neighbor(1)->index(fh)));
le.push_back(Edge(fh->neighbor(2),fh->neighbor(2)->index(fh)));

list<Edge>::iterator lit=le.begin();
list<Edge>::iterator litt;
int ih;
while(lit != le.end()){
  if ( test_conflict((*lit).first, p)){
    fh = (*lit).first;
    ih = (*lit).second;
    litt = lit; // this one has to be deleted
    lit = le.insert(lit, Edge(fh->neighbor(cw(ih)),
      fh->neighbor(cw(ih))->index(fh)));
    lit = le.insert(lit, Edge(fh->neighbor(ccw(ih)),
      fh->neighbor(ccw(ih))->index(fh)));
    le.erase(litt);
  }
  else
    ++lit;
}
if (!le.empty())
  result = true;
      }
  }
  return result;
}

//--------------------------------------------------------------------

template < class Gt, class Tds>
Natural_neighbors_2<Gt,Tds>::Natural_output
```

```
Natural_neighbors_2<Gt,Tds>::natural_areas(const Point& p, bool& in_CH, double&
suma_area) const
{
  Natural_output retour_tmp, retour;
  list<Edge> L;
  Point Vor1, Vor2, Vor3;
  bool already;
  in_CH = true;


  if  (!find_conflicts(p,L,already))
    {
      if (already)
retour.push_back(Natural_data (nearest_vertex(p), Coord_type(1)));
      else
in_CH = false;
      //    cout<<"ERROR1 !!!"<<endl;
    }
  else
    {
      list<Edge>::const_iterator L_it;
      double sum_aera(0);

      bool valid(!is_infinite(locate(p)));

      for(L_it = L.begin(); L_it != L.end(); ++L_it){
Coord_type aire(0);
Vertex_handle vert_courant =
  L_it->first->vertex(cw(L_it->second));
if (!is_infinite(vert_courant)){
  list<Edge>::const_iterator search = L.begin();
```

```
  bool test(true);
  while(test){
    if((search->first->has_vertex(vert_courant))&&(search!=L_it))
      {
if ((search->first->index(vert_courant)) != (search->second))
  test = false;
else
  ++search;
      }
    else
      ++search;
  };

  if ((!is_infinite(*L_it))&&(!collinear(segment(*L_it).source(),
 segment(*L_it).target(),p)))
    Vor1 =

geom_traits().circumcenter(segment(*L_it).source(),segment(*L_it).target(),p);
  else
    valid = false;
  Face_circulator fc = incident_faces(vert_courant,L_it->first);

  ++fc;

  if ((!is_infinite(Face_handle (fc))))
    Vor2 = dual(Face_handle (fc));
  else
    valid = false;

  Vertex_handle vert_courant_brother = search->first->vertex(cw(search->second));
```

```
  while(!(fc->has_vertex(vert_courant_brother))){
    ++fc;

    if ((!is_infinite(Face_handle (fc))))
      Vor3 = dual(Face_handle (fc));
    else
      valid = false;

    aire +=
      abs(det2x2_by_formula(Vor2.x()-Vor1.x(), Vor3.x()-Vor1.x(),
    Vor2.y()-Vor1.y(), Vor3.y()-Vor1.y())/2);
    Vor2 = Vor3;
  };

  if ((!is_infinite(*search))&&(!collinear(segment(*search).source(),
   segment(*search).target(),p)))
    Vor3 =
      geom_traits().circumcenter(segment(*search).source(),
 segment(*search).target(),p);
  else
    valid = false;

  aire +=
    abs(det2x2_by_formula(Vor2.x()-Vor1.x(), Vor3.x()-Vor1.x(),
  Vor2.y()-Vor1.y(), Vor3.y()-Vor1.y())/2);

  if(!valid)
    {
      Face_handle fh = locate(p);
      if (is_infinite(fh))
in_CH = false;
```

```
      //else
      //  cout<<"ERROR!!!!"<<endl;
    }

  sum_aera += aire;
  Natural_data sortie_courante(vert_courant, aire);
  retour_tmp.push_back(sortie_courante);
}
      };
      suma_area=sum_aera;
      //      ofstream areas_file("areas.dat");
      //      areas_file <<sum_aera<<endl;

#ifdef DEBUG
      double sum_debug_x(0);
      double sum_debug_y(0);
      std::cout << "Le point p (" << p << ") a pour voisins naturels :" << std::endl;
#endif // DEBUG

      Natural_output::const_iterator ret_it;
      for(ret_it = retour_tmp.begin(); ret_it != retour_tmp.end(); ++ret_it)
if (ret_it->second != 0)
  retour.push_back(Natural_data
   (ret_it->first, ret_it->second/sum_aera));

#ifdef DEBUG
      for(ret_it = retour.begin(); ret_it != retour.end(); ++ret_it)
{
  std::cout << ret_it->first->point() <<
    " associe au coefficient : " << ret_it->second << std::endl;
  sum_debug_x +=
```

```
    ret_it->second*(p.x()-ret_it->first->point().x());
  sum_debug_y +=
    ret_it->second*(p.y()-ret_it->first->point().y());

}
      std::cout << "La somme de controle, selon x et y, est : " << sum_debug_x
<< ", " << sum_debug_y << std::endl << std::endl;
#endif // DEBUG
    }
  return retour;
}
```

```
//--------------------------------------------------------------------
CGAL_END_NAMESPACE
//--------------------------------------------------------------------

#endif // NATURAL_NEIGHBORS_2_H
```

```
void
interpolation(Vertex_iterator vertices,Nn& dt){

/*--- Local triangulation to compute the interpolation ---*/

 //Triangulation with the vertices incident to  ''vertices[i]''

      double func_inter=0;
      Nn::Vertex_circulator circ,done;
      Nn mini_dt, nn_dt;
      circ=dt.incident_vertices(vertices);
      done=circ;

      int array_dim=0;
      do{
Vertex_handle vert_label;
vert_label=mini_dt.insert(circ->point());
vert_label->label=circ->label;
array_dim++;
circ++;
      }while(done!=circ);


  bool valid;

  int cont=1;
  double sum_area;
  double lam=0.0;
  out V = mini_dt.natural_areas(vertices->point(), valid, sum_area);
  vertices->suma_area=sum_area;
```

```
  if (valid)
    {
      out::const_iterator V_it;
      for(V_it = V.begin(); V_it != V.end(); ++V_it)
{
  Vertex_handle vert(V_it->first), vert_label;
  //triangulation only with the points that are natural neighbors
  //to compute the area of the circumcircles
  vert_label=nn_dt.insert(vert->point());
  vert_label->label=vert->label;
  //interpolation
  vertices->h_kernel[cont]=h_interpolation(V_it->second,sum_area);
  vertices->num[cont]=vert->label;
  func_inter+=vertices->h_kernel[cont];
  cont++;

  // Lambda=Sum[(r_j-r_i)^2 * W]
  lam+=squared_distance(vertices->point(),vert->point())*V_it->second;
}
    }


  // --------

  vertices->num_vecinos=cont-1;
  vertices->suma_func=func_inter;
  vertices->h_kernel[0]=-func_inter;
  vertices->num[0]=vertices->label;

  // ----------------------------------------
  //------------ Area of the circumcircles---
```

```
  // ----------------------------------------

  double area_intersec=0.0;
  double area_total=0.0;
  double radio_circum;
  // inserting vertices[i]
    Vertex_handle vert_label;
    vert_label=nn_dt.insert(vertices->point());
    vert_label->label=vertices->label;

  //iteration around the triangles
  Nn::Face_iterator fc;

  for(fc=nn_dt.faces_begin();fc!=nn_dt.faces_end();fc++)
    {
      // radio of the circumcircles
      Point p123[3];
      for (int i=0;i<3;i++)
{
  p123[i]=fc->vertex(i)->point();
}
      Circle circle(p123[0],p123[1],p123[2]);
      if(circle.is_degenerate()==0)
{
  radio_circum=circle.squared_radius ();

  for (int i=0;i<3;i++)
    {
      double c_arc=squared_distance
(vertices->point(),fc->vertex(i)->point());
  double d_arc=radio_circum - c_arc/4.0;
```

```
  area_intersec+=radio_circum * acos(pow(d_arc/radio_circum,0.5))
    -pow(d_arc,0.5)* pow(radio_circum-d_arc,0.5);
    }
  area_total+=radio_circum*M_PI-area_intersec;
}
      area_intersec=0.0;
    }

  vertices->lambda=lam;
  vertices->soporte=area_total;


  mini_dt.clear();
  nn_dt.clear();

}
```

References

[1] S. Koshizuka and Y. Oka. - Moving Particle Semi-Implicit Method for Fragmentation of Incompressible Fluid. - NUCLEAR SCIENCE AND ENGINEERING: 123, 421-434 (1996).

[2] A. F. Ghoniem and F. S. Sherman. -Grid-free Simulation of Diffusion Using Random Walk Methods.- J. of Computational Physics 61, 1-37 (1985).

[3] Strang and Fix, 1973

[4] N. Sukumar. -The Natural Element Method in Solid Mechanics-, Northwestern University, Evanston, Illinois. June 1998.

[5] Babuska and Aziz, 1976

[6] Computational Geometry Algorithm Library. Release is 2.2 (October '00). http://www.cgal.org.

[7] Visualisation Generale Interactive d'Ecoulements. Version 1.7 [24 June 1997]. http://www-sop.inria.fr/sinus/Softs/vigie.html.

[8] Jean-Daniel Boissonnat and Frédéric Cazals. - Natural Coordinates, Distance Function and Smooth Interpolation on Sampled Surfaces. - Pré-rapport de recherche I.N.R.I.A..

[9] C. Hirsch, Numerical Computation of Internal an External Flows, Jhon Wiley & Sons, New York (1988).

[10] Melenk and Babuska (1996).

[11] C. K. Chu, Adv. Appl. Mech. 18 (1978), 285.

[12] Farin, 1990.

[13] O.C.Zienkiewicz and R.L. Taylor. -The Finite Element Method-.Vol.1, 260.

Unité de recherche INRIA Sophia Antipolis

2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique

615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur

INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)

http://www.inria.fr

ISSN 0249-6399